1. Введение

В процессе компиляции программы в машинный код конкретного процессора возникает задача **распределения регистров**. Множество исследований посвящено данной проблеме, мы же рассмотрим решения, предложенные в работах Чайтина (Chaitin) [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 ещё до работ Чайтина в работах Кокке (Cocke) в 1970 [3], Ершова в 1962 [4], Шварца (Schwartz) в 1973 [6]. Второй подход — это **линейная аллокация**, которая 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)**, которое 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 размещать некоторые переменные в памяти.

Для понимания задачи рассмотрим пример: программа содержит выражение, в котором используются три переменные, в то время как архитектура процессора предусматривает лишь два регистра. В этом случае однозначно распределить переменные по регистрам не получится. Возникает необходимость проводить **выгрузку** (в англоязычной литературе *spill*) одной 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 превышает скорость работы с внешней памятью на один-два порядка.

3. Определение сложности задачи

Для того чтобы доказать, что задача является **NP-трудной**, необходимо построить полиномиальную редукцию от некоторой NP-полной задачи, а также показать, что решение данной задачи позволяет получить решение некоторой NP-полной задачи.

Сначала введем некоторые определения.

Определение 1.

Представим программу в виде CFG (control flow graph). Будем считать, что переменная x *жива* в точке p, если существует путь в CFG от точки присваивания значения переменной x, проходящий через p к точке использования переменной, на котором отсутствует другое присваивание значения переменной x.

Определение 2.

Будем говорить, что переменные a и b *зацеплены*, если одна из переменных жива в точке присваивания значения 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 проблема распределения переменных по n регистрам для данной 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)

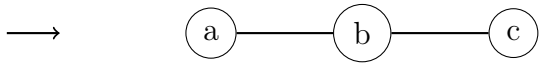


Рис. 1: Пример построения графа зацепленности
def: Объявление переменной
use: Использование переменной

В графе на рисунке 1 присутствует ребро (a, b), поскольку в точке 2 переменная a *жива*, однако ребра (a, c) в графе нет, поскольку в точке 4 для переменной a отсутствует путь, проходящий через точку её использования.

Теперь докажем, что по произвольному графу можно построить программу, распределение регистров которой даст раскраску этого графа.

Пусть V — множество переменных, R — множество регистров. Тогда существует отображение $\varphi : V \to R$, при котором выполняется следующее свойство:

$$\forall a, b \in V, \text{ если } a \text{ и } b \text{ зацеплены, то } \varphi(a) \neq \varphi(b).$$

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

В статье [2] Чайтин предложил следующий алгоритм построения кода на основе графа:

1. Если в графе существует вершина NODE_i, то в исходном коде объявляется переменная с соответствующим названием.

2. Если в графе присутствует ребро $(\text{NODE}_i, \text{NODE}_j)$, то в исходный код добавляется операция, использующая эти переменные, например суммирование, чтобы они не могли занимать один регистр.

Этот метод не всегда эффективен. В некоторых случаях требуются дополнительные конструкции, такие как *операторы ветвления*, чтобы задача раскраски графа сводилась к задаче распределения регистров. Рассмотрим пример, когда такие преобразования необходимы.

Пример 2.

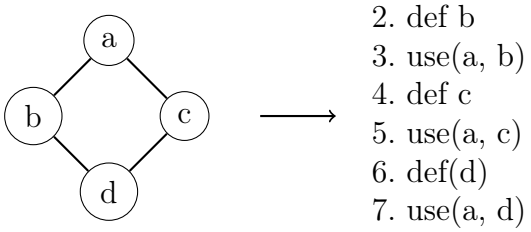


Рис. 2: Пример преобразования графа в код

На рисунке 2 показано, что код, построенный по алгоритму, предложенному Чайтином, не всегда соответствует исходному графу. В данном случае в исходном коде наблюдается зацепленность переменных a и d, хотя в исходном графе ребро (a, d) отсутствует. Код, построенный на основе графа из рисунка 2, должен соответствовать изображению 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:

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. Последовательно удалить все вершины графа, помещая их в стек (уменьшая тем самым степени других вершин) в соответствии с приведенными правилами:
   (a) Если существует вершина со степенью, меньшей N, убираем её из графа и помещаем в стек, затем переходим к пункту 2.
   (b) Если граф стал пустым, переходим к пункту 3.
   (c) В противном случае выбираем вершину со степенью, превышающей N, и убираем её (выполняем выгрузку соответствующей переменной). При этом необходимо обеспечить механизм выгрузки вершины из памяти перед её использованием и последующей загрузки обратно. После этого переходим к пункту 1.

3. Извлекать вершины из стека по одной, возвращать их в граф и присваивать каждой цвет.

Рассмотрим детали процесса. Сначала необходимо простроить граф зацепленности. Далее убираем все вершины, у которых число соседей меньше N. Важно отметить, что это действие не влияет на *хроматическое* число графа. Действительно, если у вершины меньше N соседей, всегда существует цвет, который можно ей присвоить. Это упрощает задачу, так как такие вершины можно удалить. После этого остаются только вершины с числом соседей не менее N.

Теперь необходимо выбрать одну из вершин для удаления. Принцип выбора, предложенный Чайтиным, заключается в следующем: для каждой вершины вычисляется значение, называемое стоимостью выгрузки. Сначала определяем количество объявлений и использований вершины, учитывая вес каждого объявления и использования. Предполагается, что вес использования или объявления переменной в цикле равен 10. Затем, для вычисления стоимости выгрузки вершины, вычисляем отношение *количества использований* к *степени вершины*.

При выборе вершины для выгрузки выбираем вершину с наименьшей стоимостью выгрузки.

Затем необходимо перестроить граф зацепленности, поскольку после добавления кода для выгрузки граф изменится. После этого нужно ещё раз попытаться раскрасить граф.

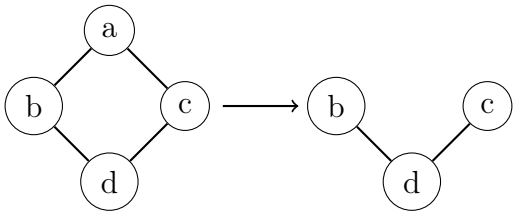
Рассмотрим, как раскрасить граф, если известно, что его можно раскрасить в N цветов. Для этого каждый раз, когда вершина удаляется из графа, будем помещать её в стек. Когда граф станет пустым, начнём извлекать вершины из стека по одной и выбирать для каждой цвет так, чтобы никакая соседняя вершина не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

Проанализируем граф, представленный на рисунке 2. Очевидно, что данный граф допускает раскраску в два цвета. Тем не менее алгоритм Чайтина не способен обнаружить такую раскраску и выгрузит одну из вершин.



Еще один пример неэффективной работы алгоритма Чайтина — это алгоритм SVD-разложения. В данном алгоритме используются несколько долгоживущих переменных, то есть интервалы жизни этих переменных включают почти весь CFG программы. Также программа содержит вложенные циклы. Проблема распределения регистров возникает из-за долгоживущих переменных. Однако, поскольку стоимость выгрузки долгоживущих переменных высока, первыми выгружаются переменные циклов. Это не решает проблему, так как она не связана с переменными циклов. В результате некоторые регистры могут оставаться свободными, несмотря на выгрузку переменных циклов в память, что дополнительно снижает эффективность работы программы.

На рисунке 4 представлена приблизительная структура кода.

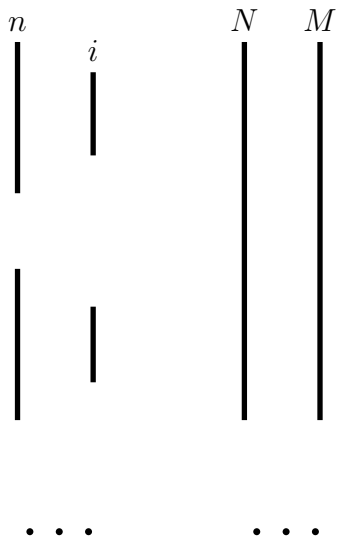


Рис. 4: Структура кода

В данном коде переменные i и n являются переменными циклов. Также присутствуют долгоживущие переменные N и M. Для упрощения можно считать, что они задают границы циклов, не участвуя непосредственно в самих циклах.

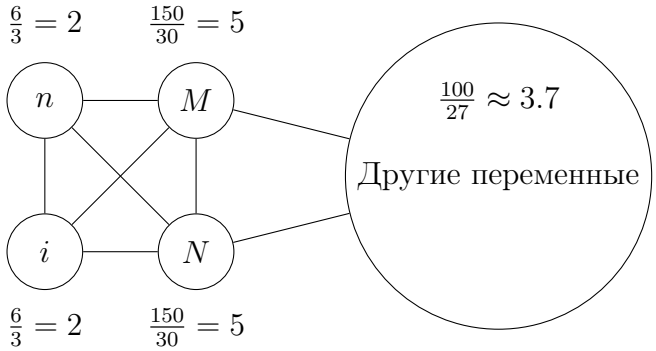


Рис. 5: Граф зацепленности и стоимости выгрузки

Предположим, что в графе имеются дополнительные переменные в количестве 27, каждая из которых используется 100 раз. Переменные i и n имеют по 3 соседа и используются 6 раз. Переменные N и M имеют по 30 соседей, и пусть каждая из них используется 150 раз. Будем считать что граф необходимо покрасить в 2 цвета. Рассчитаем стоимость выгрузки для каждой вершины: $\text{cost}(i) = \text{cost}(n) = \frac{6}{3} = 2$, $\text{cost}(N) = \text{cost}(M) = \frac{150}{30} = 5$, $\text{cost}(\text{other}) = \frac{100}{27} \approx 3.7$. Как следует из расчетов, первыми будут выгружены переменные i или n, так как их стоимость выгрузки минимальна.

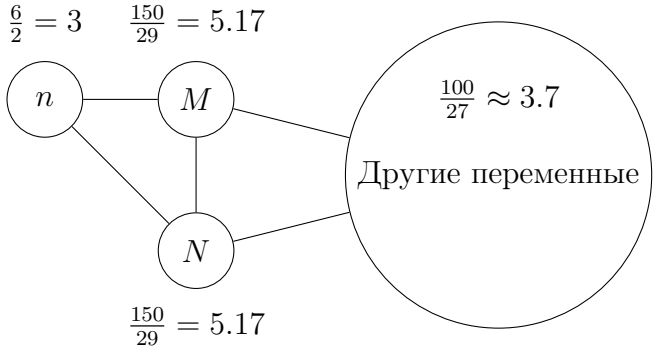


Рис. 6: Пересчитанные стоимости

Предположим, что переменная i была выгружена. После этого в графе всё ещё остаются переменные, имеющие более двух соседей, поэтому согласно пункту 2c алгоритма необходимо выбрать следующую переменную для выгрузки. Пересчитаем стоимости выгрузки (см. Рисунок 6). Алгоритм предлагает выгрузить переменную n, переменную цикла, однако это не решит проблему. В итоге 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**. Разберём какие именно проблемы скрываются в вышеописанных 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 нет вершин со степенью меньшей N, то такой граф нельзя раскрасить в N цветов.

В примере с SVD-разложением проблема возникает не когда алгоритм выбирает выгрузить переменные i и n, а когда после выгрузки N и M он не обнаруживает, что переменные n и i могут поместиться на регистры. Во время этапа 3 алгоритм не рассматривает причину выгрузки переменных i и n. Он не понимает, что причиной для выгрузки i и n были I и N. Однако переменные I и N были выгружены, а значит, i и n могут быть размещены на регистрах.

5.2. Идея

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

1. В пункте 2c алгоритма, когда все оставшиеся вершины имеют не менее N соседей, по тому же принципу выберем переменную для выгрузки, однако вместо немедленной выгрузки, разместим ее на стек.

2. Соответственно, в пункте 3 возникает проблема, ведь теперь не всегда вершину можно будет покрасить. Если оказалось, что для вершины нельзя подобрать цвет, оставим ее неокрашенной. Непокрашенные переменные — это переменные, которые нужно выгрузить.

Эти изменения помогут решить обнаруженные проблемы алгоритма Чайтина. Первая проблема решается в пункте 1. Теперь в примере, представленном на рисунке 2, хотя некоторая вершина и будет выбрана для выгрузки, на этапе 3 всем вершинам удастся получить цвет.

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

Эвристика предыдущего алгоритма скорее отвечала на вопрос “имеет ли вершина менее N соседей?”, чем на вопрос “можно ли подобрать цвет для этой вершины?”. Новый алгоритм стремится ответить 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. Он 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 согласно результатам, представленным в статье, производительность кода, сгенерированного с использованием этого алгоритма, снижается незначительно.

6.1. Идея

Представим программу в некотором IR и пронумеруем его операции произвольным образом. Например, в качестве тривиального подхода подойдет нумерация строк исходного кода программы. Альтернативно, можно пронумеровать линейные участки в CFG, например в порядке обхода DFS.

Определение 4. *Интервалом жизни* переменной v будем называть отрезок $[i, j]$, где i — это наименьший номер точки в которой переменная жива, а j — наибольший. Интервалы жизни для переменной a будем обозначать interval_a.

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

Заметим, что в интервале жизни переменной могут встречаться точки, в которых она не жива.

Пример 3.

Рассмотрим пример с рисунка 7. При выбранной нумерации IR интервал жизни переменной $a = [1, 5]$, однако в точке 3 переменная не жива, так как если исполнение пошло по этому пути то переменная больше не будет использована.

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

Рис. 7: Псевдокод

Хотя алгоритм не опирается на какую-то конкретную нумерацию, выбор нумерации может значительно повлиять на результат аллокации. Например, если необходимо произвести аллокацию быстро, можно пронумеровать линейные участки CFG. Однако при таком методе нумерации не будут учтены операции внутри линейных участков. С другой стороны можно пронумеровать каждую операцию над переменными. В таком случае будет играть важную роль то, как они упорядочены. От этого и будет зависеть количество мертвых точек внутри интервалов жизни. В общем случае лучшей нумерацией будет топологическая сортировка.

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 меняется только если какая-нибудь переменная станет живой в интервальном смысле или, наоборот, какая-нибудь перестанет быть живой. Чтобы выбрать переменные для выгрузки, используется эвристика, при которой отдается предпочтение переменной с более поздним окончанием интервала жизни.

Алгоритм, предложенный Полетто:

1. Удалить переменные, умирающие в точке p, из списка живых. Это переменные, чьи интервалы жизни которых заканчиваются на p.

2. Добавить оживающие переменные в список живых. То есть те, интервалы жизни которых начинаются в точке p. На этом этапе может возникнуть 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

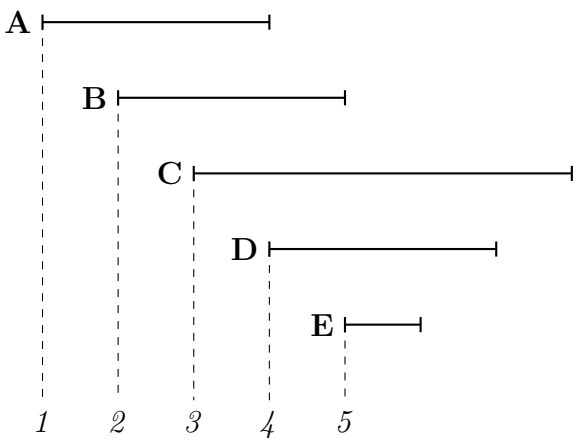


Рис. 8: Пример интервалов жизни

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 те переменные, которые станут живыми в текущий момент времени и, возможно, одну переменную, которая оживает потом.

Пример 4.

Рассмотрим пример работы алгоритма линейной аллокации. На рисунке 8 представлены интервалы жизни некоторых переменных. Пусть в данном случае необходимо разместить переменные в двух 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 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, а 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, проведённое в статье Полетто [5]. Сравниваются как различия во времени компиляции программы, так и влияние на производительность во время выполнения.

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 подсчёта количества использований. Этот подход выходит за рамки реферата, и мы не будем его комментировать. 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.

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 всё же быстрее для линейной аллокации.

7.2. Время исполнения

На рисунке 10 представлены результаты тестирования программ (меньше — лучше), скомпилированных при помощи различных алгоритмов аллокации регистров. В таблице представлены четыре алгоритма, нас интересуют только Linear scan и Graph coloring.

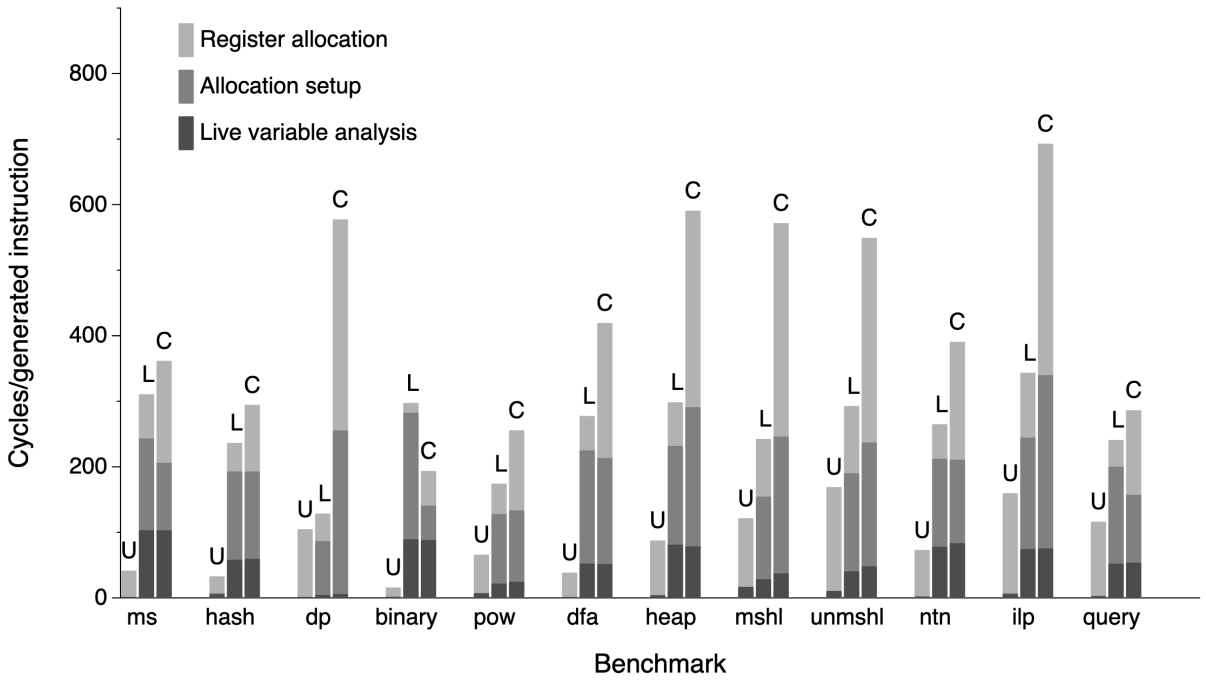


Рис. 9: Время компиляции

Значения в скобках показывают отношение времени выполнения программы, скомпилированной с данным алгоритмом, ко времени выполнения программы, скомпилированной с использованием алгоритма раскраски графа.

Результаты показывают, что время выполнения программ, скомпилированных с применением линейной аллокации, отличается от результатов алгоритма раскраски графа в пределах 10%.

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

8. Заключение

Как следует из приведённых выше сравнений, линейная аллокация выполняется быстрее, чем алгоритмы, основанные на раскраске графов. Однако скорость исполнения программ, скомпилированных с использованием этого алгоритма, незначительно отличается от скорости программ, скомпилированных с использованием алгоритмов, основанных на раскраске графов.

Кроме того, сам алгоритм значительно проще в реализации.

Совокупность этих факторов делает его привлекательной альтернативой стандартным методам аллокации регистров там, где важна скорость компиляции. Примером таких областей применения является, например, JIT-компиляция.

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.