# Быстрый алгоритм минимизации высоты графа зависимостей

В.Ю. Волконский, В.Д. Гимпельсон, Д.М. Масленников

**Аннотация.** В работе представлен быстрый алгоритм разрыва зависимостей на предикатном коде для архитектур с явно выраженным параллелизмом. Алгоритм имеет линейную сложность по числу возможных разрывов и, благодаря этому свойству, может быть использован в составе динамического оптимизирующего транслятора.

## Введение

Динамическая оптимизирующая двоичная трансляция является перспективной, бурно развивающейся отраслью вычислительной технологии [5, 9, 16]. Интерес к ней в настоящее время резко возрос благодаря появлению архитектур нового поколения и необходимост решения для них проблем совместимости с архитектурой x86, являющейся на настоящий момент доминирующей. Помимо требований по качеству оптимизированного кода к динамическому компилятору предъявляются жесткие требования по затратам времени на выполнение оптимизаций.

Одними из основных представителей новых архитектур являются микропроцессоры, основанные на EPIC архитектуре. Примерами могут служить микропроцессоры семейства *"Itanium"* фирмы Intel [7, 8] и микропроцессоры *"Cruso"* и *"Astro"* фирмы Transmeta [15], для которых также существуют двоично-оптимизирующие комплексы IA-32 Execution Layer [9] и Transmeta Code Mmorphing Software [16], соответственно. Характерной особенностью EPIC архитектур является одновременное исполнение нескольких семантически независимых операций за такт. В связи с этим в технологии программной оптимизации существенно возрастает значение метода, обычно характеризуемого как "разрыв зависимости" между операциями. В формальном аспекте его можно рассматривать как метод, направленный на сведение к минимуму высоты графа зависимостей между операциями.

В работе предложен алгоритм минимизации высоты графа зависимостей, имеющий линейную сложность по числу возможных разрывов, согласно которому разрываются не все зависимости, а только некоторое их подмножество. Он был реализован в двоичном оптимизирующем компиляторе, разработанном для архитектуры "Эльбрус" [6].

## 1. Постановка задачи и основные определения

Имеется некоторое промежуточное представление семантики, основными элементами которого являются операции. Над операциями можно построить граф зависимостей.

**Определение 1.** Граф зависимостей – это ориентированный связный граф без циклов, узлами которого являются все операции промежуточного представления кода. Дуга соединяет две операции тогда и только тогда, когда операция – предшественник дуги по некоторой причине должна выполниться не позже операции – преемника дуги.

Заметим, что в данной работе исследуются только ацикличные графы, хотя, в принципе, возможно также рассматривать циклический граф зависимостей.

Все причины появления зависимостей между операциями определяются алгоритмом построения графа зависимостей, который фактически анализирует каждую пару операций на наличие между ними зависимости того или иного типа. Самые распространенные причины упорядочивания операций – потоковые зависимости (использование результата одной операции в качестве операнда другой), антизависимости (переназначение ресурса, занимаемого операндом одной операции, под результат другой), output-зависимости или зависимость по результату (переназначение ресурса, отданного под результат одной операции, под результат другой), зависимости между обращениями в память и т.д. [20]. Необходимо также обратить внимание на особый тип потоковой зависимости – предикатную зависимость. В EPIC-архитектурах существуют особые предикатные регистры. На операцию может действовать один из предикатов, и это означает, что операция будет исполняться тогда и только тогда, когда соответствующий предикат истинен. Подробнее этот вопрос рассматривается в [7] и [8]. Кроме того, в определенных контекстах появляются специфические типы и причины зависимостей между операциями. Так, например, специфическими для двоичной компиляции являются зависимости между всеми операциями изменения внешнего контекста (записи в память) и всеми операциями, являющимися причинами потенциальных прерываний.

Граф зависимостей является взвешенным графом. Каждая дуга графа зависимостей имеет длину, которая определяет, через сколько тактов после выполнения предшественника дуги может начаться выполнение последователя дуги. Например, большинство операций целочисленной арифметики обычно исполняются за один такт, а операции плавающей арифметики за несколько тактов. Так как граф зависимостей является ациклическим, найдётся хотя бы одна вершина, в которую не входит ни одна дуга, и хотя бы одна вершина, из которой не выходит ни одной дуги. Добавим к графу две вершины, называемые `ENTER` и `END`. Построим дуги, соединяющие `ENTER` со всеми вершинами, в которые не входит дуга, а также дуги, соединяющие все вершины, из которых не выходят дуги, с узлом `END`. В дальнейшем будем рассматривать только такие расширенные графы зависимости. Это корректно, так как приведённые преобразования не нарушают связности и ацикличности.

**Определение 2**. Высотой графа зависимостей называется максимальная длина пути, ведущего от `ENTER`-а к `END`-у.

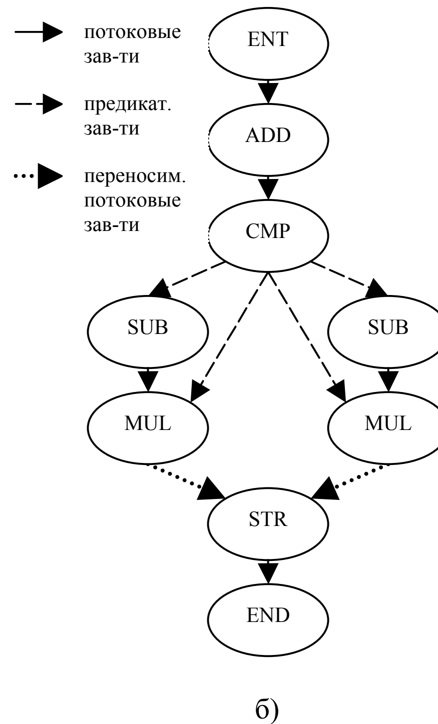
Фактически высота графа зависимостей есть минимальная длительность исполнения данного фрагмента кода микропроцессором заданной архитектуры, сократить которую уже невозможно. Однако, если ширина командного слова микропроцессора достаточно велика (то есть может выполняться достаточно много команд за такт), то в большинстве случаев высота графа зависимостей и будет характеризовать то время, за которое исполнится данный код. В связи с этим, минимизация высоты графа зависимостей является важной задачей.

Существует много различных способов решения этой задачи на скалярном коде [1-4,17-19]. Описанный в работе алгоритм использовался для разрыва потоковых предикатных зависимостей, антизависимостей и зависимостей между записями и чтениями в память. Однако сфера его применения не ограничивается разрывами только этих трёх типов зависимостей. Он может использоваться для разрыва других зависимостей при условии, что преобразование, разрывающее зависимость, удовлетворяет определённым условиям (они описаны в главе 2). Например, с помощью этого алгоритма можно реализовать оптимизацию unzipping [19]. Приведём пример разрыва потоковой предикатной зависимости. На Рис.1 изображено промежуточное представление и соответствующий ему граф зависимостей.

Поясним запись промежуточного представления. Команда задает название операции, её аргументы (константу, содержимое адресуемого регистра) и справа от стрелки указывает регистр, в который помещается результат. В команде может задаваться условие, например, `IF(P0=TRUE) SUB r1 2→ r5` означает, что данная команда выполнится тогда, и только тогда, когда предикат `P0` истинен.

```
    ENTER
ADD  r0  1   →  r0
CMP  r0  0   →  P0
IF(P0=TRUE)  SUB  r1  2   →  r5
IF(P0=FALSE) SUB  r1  4   →  r5
IF(P0=TRUE)  MUL  r5  2   →  r6
IF(P0=FALSE) MUL  r5  3   →  r6
ST   r6  0  23
END
```

а)




б)

Рис.1. Граф зависимостей для промежуточного предикатного представления

а) Последовательность операций промежуточного представления
б) Граф зависимостей

Пусть операции ADD (сложение), SUB (вычитание), ST (запись в память) выполняются за один такт, операция CMP (сравнение двух значений с выработкой предикатного регистра) – за два такта, а операция MUL (умножение) – за три такта. Тогда высота графа зависимостей будет равна восьми тактам. Теперь разорвём потоковые предикатные зависимости, идущие от операции CMP (сравнение двух значений с выработкой предиката). Преобразованное представление и граф зависимостей представлены на Рис. 2.

После этих преобразований высота графа зависимостей стала равна шести, однако появилось две новых однотактных операции MOV (пересылки значения). В приведенном примере разрываются потоковые предикатные зависимости для двух операций вычитания и для двух операций умножения и тем самым уменьшается общая высота графа зависимостей. Здесь демонстрируются две типичные для разрыва зависимостей ситуации. Во-первых, уже в этом простом случае видно, что алгоритм минимизации высоты графа зависимостей должен быть глобальным – он должен анализировать все зависимости, являющиеся кандидатами на разрыв. Так, в примере разрыв входных предикатных потоковых зависимостей только для одной из операций умножения не привел бы к достижению минимальной высоты графа.

Во-вторых, для разрыва зависимостей использовались вновь введенные операции. Фактически высота графа была уменьшена за счет увеличения нагрузки на вычислительные ресурсы (исполнительные устройства процессора). Однако ширина командного слова реальных микропроцессоров ограничена, поэтому разрыв всех возможных зависимостей может породить слишком большое количество новых операций, что при их планировании в условиях ограниченных ресурсов приведет к удлинению критического пути. Таким образом, необходим алгоритм, который будет разрывать зависимости экономно.

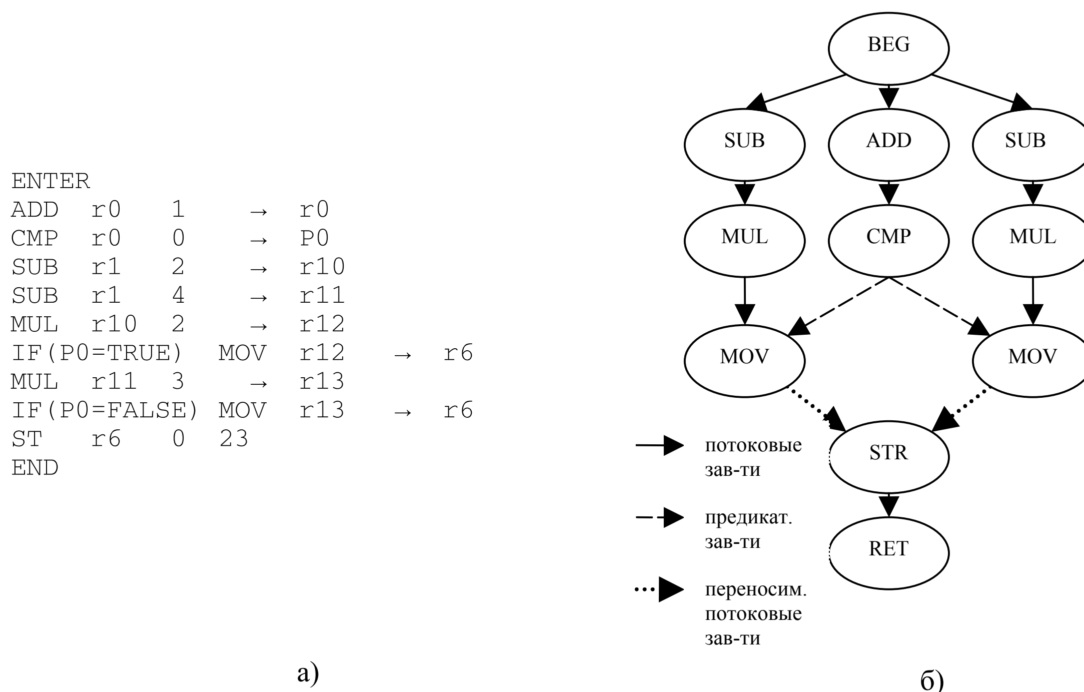



Рис.2. Граф зависимостей для промежуточного предикатного представления после разрыва зависимостей

а) Последовательность операций промежуточного представления

б) Граф зависимостей

## 2. Формализация задачи

Формализуем задачу в терминах теории графов. Пусть имеется произвольный граф зависимостей. Есть некоторое число зависимостей, которые можно разорвать (Рис.1 *б)* они проведены пунктирной линией). Заключение о том, что зависимость может быть разорвана, вытекает из ее типа (то есть причина, по которой строилась зависимость, например, можно разрывать потоковые предикатные зависимости).

Некоторые зависимости могут быть разорваны с помощью переименования, и для их разрыва не требуется построение нового узла. Это справедливо для зависимости между операциями CMP и SUB на Рис.1. У операции SUB можно убрать предикат, но при этом необходимо заменить регистр, заданный в её поле результата, и в тех операциях, где используется результат (то есть произвести переименование).

Разрыв зависимостей с помощью переименования можно осуществлять всегда, не беспокоясь о том, что это может привести к увеличению нагрузки на исполняющие устройства, поскольку не создаётся новых операций. В силу сказанного, без ограничения общности можно считать, что все разрывы зависимостей, возможные с помощью переименования, уже осуществлены. Рассмотрим только те случаи, для которых необходимо построение нового узла.

Пусть задан узел $A$ с входящей зависимостью, которую можно разорвать. Построим новый узел $A'$. Все дуги, входящие и выходящие из узла $A$, распределяются между узлами $A$ и $A'$ следующим образом. Все дуги, входящие в $A$, которые помечены как разрываемые, переносятся на новый узел. Остальные дуги, которые входят в $A$, никуда не перемещаются и остаются входными дугами этого узла. Множество дуг, выходящих из узла $A$, разделятся на два подмножества: дуги, которые после разрыва зависимости попрежнему выходят из узла $A$, и дуги, которые после разрыва переносятся на узел $A'$. Разделение дуг на эти два подмножества зависит от типа разрываемой зависимости, и однозначно определяется им. Фактически, это разделение на подмножества следует вы-

полнить таким образом, чтобы получаемое представление было корректным (то есть чтобы старое и новое представления были эквивалентны с точки зрения реализуемых функций). Например, в случае разрыва потоковой предикатной зависимости переносимыми являются выходящие потоковые дуги такие, что последователь этой дуги может использовать переносимое дугой значение не только из узла – предшественника дуги, но и ещё от какого-нибудь узла. Например, на Рис.1 операция ST может использовать значение от обеих операций MUL, хотя заранее не известно - от какой именно. Все остальные выходящие дуги при разрыве потоковой предикатной зависимости не переносятся. На Рис.1 переносимые дуги нарисованы точками.

Отметим, что при разрыве зависимости необходимо построить дугу с длиной, равной 1, между узлами $A$ и $A'$. На Рис.1 такие дуги входят в операции MOV.

Для формулировки алгоритма понадобится ввести несколько определений.

**Определение 3.** Временем раннего планирования узла графа зависимостей называется максимальная из длин всех путей, ведущих от узла ENTER к данному узлу.

**Определение 4.** Временем позднего планирования узла графа зависимостей называется величина, равная времени раннего планирования узла END минус максимальная из длин всех путей, ведущих от данного узла к узлу END.

Нетрудно заметить, что время раннего планирования узла END, равное его времени позднего планирования, и есть высота графа зависимостей.

Определение 5. Критическим путём в графе зависимостей называется путь от ENTER к END такой, что у всех узлов в этом пути времена раннего и позднего планирования совпадают.

Определение 6. Дуга $v$ называется определяющей для узла $A$, если

$$early(pred(v)) + delay(v) > \max_{\substack{succ(u)=A \\ u \neq v}} \big(early(pred(u)) + delay(u)\big),$$

где $pred(\cdot)$ – предшественник дуги, $succ(\cdot)$ – последователь дуги, $early(\cdot)$ – время раннего планирования узла, $delay(\cdot)$ – длина дуги.

Эта формулировка означает, что фактически время раннего планирования узла задается его определяющей дугой.

## 3. Известные алгоритмы

Известны два класса алгоритмов решения этой задачи. Первый класс – переборные алгоритмы, которые приводят исходный граф зависимостей к минимальной высоте, строя наименьшее возможное количество новых узлов.

**Алгоритм 1.** Возьмём множество всех зависимостей, которые можно разорвать. Выберем все подмножества этого множества, и для каждого подмножества разорвём все входящие в него зависимости. После этого определим высоту графа зависимостей. В заключение найдем среди подмножеств, дающих минимальную высоту графа зависимостей, подмножество минимальной мощности, а если таких подмножеств несколько, то выберем из них произвольное. Это и будет минимальное подмножество тех зависимостей, которые надо разорвать для достижения минимальной высоты графа зависимостей.

Другое формальное описание такого переборного алгоритма приведено в [1,2]. В этих работах проблема минимизации высоты графа зависимостей сводится к решению задачи целочисленного линейного программирования. Назовём такое решение **Алгоритм 1'.**

Ко второму классу относятся итеративные алгоритмы. Приведем простой итеративный алгоритм, который сводит к минимуму высоту графа зависимостей, возможно, используя при этом не минимальное количества новых узлов.

**Алгоритм 2**. Производится разметка раннего и позднего времени планирования. Находится множество узлов F со следующими свойствами:

- все узлы лежат на критическом пути;
- к каждому узлу идёт дуга, которую можно разорвать, причём эта дуга является определяющей;
- на каждом критическом пути находится по крайней мере один узел из множества F.

Такое множество узлов назовём "фронтом". Если фронт не найден, то граф уже имеет минимально возможную высоту, так как есть критический путь, лишенный дуг, которые можно разорвать. Этот путь и определяет высоту графа зависимостей. Если фронт найден, то разрываются все зависимости, идущие к узлам из фронта, и выполняется переход к началу алгоритма. Фронт находится следующим образом:

1. В качестве текущего узла выбирается `ENTER`.

2. Текущий узел помечается как обработанный. Далее, обходятся все дуги, выходящие из текущего узла. Пусть последователь текущей дуги обозначен через $C$.

Если узел $C$ не находится на критическом пути или помечен как обработанный, то переходим к следующей дуге.

Если текущая дуга является определяющей для узла $C$ и её можно разорвать, то добавляем узел $C$ во фронт и переходим к следующей дуге, иначе добавляем узел $C$ в список необработанных узлов.

Если через $C$ обозначен узел `END`, то фронт не найден и процесс завершается (найден критический путь, лишенный определяющих дуг, которые можно разорвать).

3. Берётся первый элемент из списка необработанных узлов.

Если такой элемент присутствует, то он принимается как текущий и выполняется переход к пункту 2, иначе процесс завершается.

В главе 5 показано, что этим алгоритмам свойственны недостатки, которые ставят под сомнение целесообразность их применения в промышленных компиляторах. В связи с этим авторами был разработан новый алгоритм разрыва зависимостей, который рассматривается в следующей главе.

## 4. Быстрый алгоритм разрыва зависимостей

Для описания разработанного алгоритма разрыва необходимо ввести понятие топологической сортировки.

**Определение 7.** Топологической сортировкой узлов графа называется такая нумерация узлов, при которой для любой дуги номер её предшественника меньше номера её последователя.

Алгоритм топологической сортировки описан в [21]. Везде далее будет рассматриваться его частный случай, предполагающий выполнение дополнительного свойства для нумерации: если в графе разорваны некоторые зависимости, то новый узел, построенный для разрыва данной зависимости, должен иметь номер на единицу больший, чем узел, к которому была направлена эта зависимость. С таким ограничением топологическая сортировка возможна, так как между этой парой узлов есть только один путь, а именно – дуга между ними.

Ниже следует непосредственная формулировка рассматриваемого алгоритма.

**Алгоритм 3.** Суть предлагаемого алгоритма состоит в том, что сначала разрываются все зависимости, которые можно разорвать, вслед за чем восстанавливаются зависимости, разрывы которых неэффективны, то есть те, которые заведомо не уменьшают высоту графа зависимостей.

Для формулировки алгоритма существенна проблема разбиения дуг на классы. Как отмечалось в главе 2, способ выполнения этого разбиения определяется причиной, по которой строилась данная зависимость. Отсюда следует, что разбиение проще всего сделать при построении графа зависимостей, так как в процессе его работы известен тип каждой построенной зависимости и можно сохранить о нем информацию. Сложность подобного разбиения на классы пропорциональна коли-

честву дуг в графе зависимостей. В дальнейшем описании алгоритма предполагается, что граф зависимостей уже построен и разметка дуг на классы задана.

Разработанный алгоритм разрыва зависимостей реализуется в четыре этапа, последовательно реализуемых функциями

```
Break_Dependence( );
Mark_Early_Time_With_Partial_Recover_Dependence( );
Mark_Late_Time( );
Final_Recover_Dependence( ).
```

Ниже с комментариями приводится полный набор функций, занятых в реализации алгоритма.

1. Функция `Break_Dependence( )` разрывает все зависимости, которые можно разорвать, и запоминает их. Её действия поясняются текстом

```
Break_Dependence(node)
{
  запоминается, что к узлу node шла зависимость, которую разорвали;
  new_node = создаётся_новый_узел;
  for edge in все предшественники node
  {
    if(edge – разрываемая зависимость)
    {
      предшествинник(edge) = new_node;
    }
  }
  for edge in все последователи node
  {
    if(edge – переносимая дуга)
    {
      последователь(edge) = new_node;
    }
  }
}
```

2. Функция `Mark_Early_Time_With_Partial_Recover_Dependence( )` производит разметку времени раннего планирования для всех узлов графа зависимостей с одновременным восстановлением части неэффективно разорванных зависимостей, а именно тех зависимостей, которые не являлись определяющими. Она также обеспечивает неувеличение высоты графа зависимостей. Увеличение может произойти, если время раннего планирования последователя разорванной зависимости не изменится, что и означает – разорванная зависимость не была определяющей. (Отметим, что в общем случае увеличение высоты графа зависимостей на один такт – длину операции `MOV` – может быть следствием построения нового узла для разрыва зависимости).

```
Mark_Early_Time_With_Partial_Recover_Dependence( )
{
  early_time(begin) = 0;
  for node в графе в порядке топологической нумерации
  {
    max_time = 0;
    for edge in все дуги входящие в node
    {
      pred_node = предшествинник(edge);
      max_time = max(max_time,время_раннего(pred_node)+длина(edge));
    }
    время_раннего(node) = max_time;
```

```
    if (к node не шла зависимость, которую разорвали
        в функ. мBreak_Dependenceo)
    {
      continue;
    }
    /* к node шла зависимость, которую разорвали. */
    /* получаем операцию от которой шла зависимость */
    dep_pred = предшественник_разорванной_зависимсти( node);
    /* получаем саму зависимость */
    dep_edge = разорванная_дуга(node);
    early_dep = время_раннего(dep_pred)+длина(dep_edge);
    if (early_dep > max_time)
    {
      continue;
    }else
    {
      /* восстанавливаем зависимость dep_edge */
      Rec_One_Dep(dep_edge);
    }
  }
}
```

3. Функция `Mark_Late_Time( )` производит разметку времён позднего планирования.

```
Mark_Late_Time( )
{
  время_позднего(end) = время_раннего(end);
  for node в графе в порядке обратной топологической нумерации
  {
    min_time = 0x7fffffff; /* максимальное целое */
    for edge in все дуги выходящие из node
    {
      succ_node = последователь(edge);
      min_time = min(min_time, время_позднего(succ_node)+длина(edge));
    }
    время_позднего(node)= min_time;
  }
}
```

4. Функция `Final_Recover_Dependence( )` восстанавливает неэффективные разрывы, оставшиеся невосстановленными в процедуре `Mark_Early_Time_With_Partial_Recover_Dependence`. Зависимость восстанавливается, если время раннего планирования нового узла не больше времени позднего планирования узла, к которому шла разорванная зависимость.

```
Final_Recover_Dependence( )
{
  for node in преемники разорванных зависимостей в порядке
              топологической нумерации,
      edge in разорванная зависимость
  {
    new_node = новый узел построенный для разрыва зависимости;
    if (время_позднего(node) ≥ время_раннего(new_node))
```

```
    {
      /* восстанавливаем зависимость */
      Rec_One_Dep(edge);
      /* необходимо скорректировать времена после
         восстановления зависимости */
     Correct_Times(node,new_node);
    }
  }
}
```

5. Функция `Correct_Times(node,new_node)` выполняет коррекцию времён раннего планирования после восстановления зависимости в функции `Final_Recover_Dependence`.

```
Correct_Times(node,new_node)
{
  /* корректируются времена раннего планирования */
  delta = время_раннего(new_node)-время_раннего(node);
 node включается в список с номером delta;
  while (есть хотя бы один элемент, хотя бы в одном списке)
  {
    m =  номер максимально по номеру не пустого списка;
    cur_node = первый_элемент_списка_с_номером_m;
    увеличивается время раннего у cur_node;
    for ( succ_cur_node in все последователи cur_node)
    {
      delta_new = для succ_cur_node вычисляется изменение
                  времени раннего;
      list_num = список,_в_котором_находится(succ_cur_node)
      if (list_num == неопределенный_список)
      {
        /* нет ни к каком списке */
        succ_cur_node включается в список с номером delta_new;
      } else if  ( list_num < delta_new)
      {
        succ_cur_node удаляется из списка с номером list_num;
       succ_cur_node добавляется в список с номером delta_new;
      }
    }
  }
}
```

6. Функция `Rec_One_Dep(edge)` выполняет восстановление одной зависимости, а именно - удаляет новый узел, построенный для разрыва зависимости, и переносит все дуги, входящие и выходящие из него, на текущий узел.

```
Rec_One_Dep(edge)
{
  node = узел_к_которому_шла_зависимость(edge);
  new_node = узел_построенный_для_разрыва(edge);
  удаляется дуга между node и new_node;
  все предшествинники и все последователи new_node переносятся на
  node;
  удаляется new_node;
}
```

Теперь можно показать, что данный алгоритм действительно формирует граф зависимостей минимальной высоты (при условии, что применяются только фиксированные типы разрыва зависимостей, которые заданы первоначальной разметкой). Сначала докажем, что механизм коррекции времени планирования, приведённый в п. 5, верен.

**Утверждение 1.** Приведенный механизм коррекции времени планирования правильно корректирует времена для узлов с топологическими номерами большими, чем $A$.

*Доказательство*. При восстановлении зависимости функцией `Final_Recover_Dependence( )` у всех узлов с большим топологическим номером время позднего планирования не изменяется. Действительно, время позднего планирования определяется длиной максимального пути, ведущего к `END`, а для узла с большим топологическим номером этот путь не проходит ни через узел $A$, ни через новый узел $A'$, построенный для разрыва зависимости, (вспомним, что мы расширили понятие топологической сортировки и узел $A$ имеет номер на единицу меньший, чем узел $A'$).

Заметим, что если у некоторого узла должно измениться время раннего планирования на величину $\varepsilon$, то у одного из предшественников этого узла это время должно измениться не меньше, чем на $\varepsilon$. Отсюда по индукции получаем, что у всех узлов с большим номером в топологической нумерации будет скорректировано время раннего планирования. Утверждение доказано.

**Теорема**. Приведенный алгоритм преобразует произвольный граф зависимостей $G$ в граф зависимостей с минимально возможной высотой для этих преобразований.

*Доказательство.* На первом этапе алгоритма в функции `Break_Dependence` разрываются все зависимости. Покажем, что при восстановлении зависимостей в соответствии с алгоритмом все разрывы, которые увеличивают высоту, будут восстановлены, а те, которые могут уменьшить высоту, останутся разорванными.

Первое восстановление зависимостей происходит при исполнении функции `Mark_Early_Time_With_Partial_Recover_Dependence`. Разделим множество всех путей в графе на два подмножества: подмножество $X$ путей, проходящих через узел $A$ (к которому шла разрываемая зависимость) или через узел $A'$ ( новый построенный узел для разрыва зависимости), и подмножество $Y$ всех остальных путей. Высота графа зависимостей по определению равна:

$$h(G) = \|X \cup Y\| = \max(\|X\|, \|Y\|),$$

где $\|M\| = \{\max|\Gamma| : \Gamma \in M - \text{множество путей}\}$. В результате наших преобразований величина $\|Y\|$ не изменяется.

Исследуем величину $\|X\|$.

Пусть $X_1$ - множество всех путей, идущих из `ENTER` в узел $A$,

$X_1'$ - множество всех путей, идущих из `ENTER` в узел $A'$,

$X_2$ - множество всех путей, идущих из узла $A$ к `END`,

$X_2'$ - множество всех путей, идущих из узла $A'$ к `END`.

Исключим из множеств $X_1'$ и $X_2'$ пути, включающие дугу, идущую от $A$ к $A'$. Для высоты графа с разорванной зависимостью имеем неравенство:

$$\|X\| = \max\left(\|X_1\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + 1 + \|X_2'\|\right).$$

Обозначим через $\overline{X}$ соответствующую величину с неразорванной зависимостью, тогда

$$\|\overline{X}\| = \max\left(\|X_1\|, \|X_1'\|\right) + \max\left(\|X_2\|, \|X_2'\|\right).$$

Рассмотрим ветвь алгоритма в случае $early_p > maxearly$. Это означает, что $\|X_1\| < \|X_1'\|$, то есть

$$\|X\| = \max(\|X_1\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + 1 + \|X_2'\|) \le$$
$$\le \max(\|X_1'\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + 1 + \|X_2'\|) \le$$
$$\le \max(\|X_1'\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1'\| + \|X_2'\|) = \|X_1'\| + \max(\|X_2\|, \|X_2'\|, \|X_2'\|) =$$
$$= \|X_1'\| + \max(\|X_2\|, \|X_2'\|) = \max(\|X_1\|, \|X_1'\|) + \max(\|X_2\|, \|X_2'\|) = \|\overline{X}\|.$$

Таким образом, при разрыве данной зависимости высота графа не увеличилась.

Теперь рассмотрим случай, когда $early_p \le maxearly$, это означает, что $\|X_1'\| \le \|X_1\|$, тогда

$$\|\overline{X}\| = \max(\|X_1\|, \|X_1'\|) + \max(\|X_2\|, \|X_2'\|) = \|X_1\| + \max(\|X_2\|, \|X_2'\|) =$$
$$= \max(\|X_1\| + \|X_2\|, \|X_1\| + \|X_2'\|) = \max(\|X_1\| + \|X_2\|, \|X_1\| + \|X_2'\|, \|X_1\| + \|X_2'\|) =$$
$$= \max(\|X_1\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + \|X_2'\|) \le$$
$$\le \max(\|X_1\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + \|X_2'\| + 1) = \|X\|.$$

Следовательно, восстановление этой зависимости не увеличивает высоту графа зависимостей. Таким образом, восстановление зависимостей в функции `Mark_Early_Time_With_Partial_Recover_Dependence`, с одной стороны, не восстанавливает зависимости, разрыв которых приводит к уменьшению высоты, с другой стороны, восстанавливает часть зависимостей, которые не уменьшают высоту, причем восстанавливаются все зависимости, которые увеличивают высоту.

Покажем теперь, что восстановления зависимостей в функции `Final_Recover_ Dependence` не увеличивают высоту графа. (Для всех зависимостей, разрыв которых увеличивает высоту, то есть восстановление уменьшает высоту, восстановление уже выполнено в функции `Mark_Early_Time_With_Partial_Recover_Dependence`).

Из условия $lateA \ge earlyA'$ следует, что $\|X\| - \|X_2\| \ge \|X_1'\| \Rightarrow \|X\| \ge \|X_2\| + \|X_1'\|$. Условие $\|X_1\| < \|X_1'\|$ также выполнено, ибо в противном случае разрыв этой зависимости был бы восстановлен в функции `Mark_Early_Time_With_Partial_Recover_Dependence`.

Итак:

$$\|\overline{X}\| = \max(\|X_1\|, \|X_1'\|) + \max(\|X_2\|, \|X_2'\|) = \|X_1'\| + \max(\|X_2\|, \|X_2'\|) =$$
$$= \max(\|X_1'\| + \|X_2\|, \|X_1'\| + \|X_2'\|) = \max(\|X_1'\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1'\| + \|X_2'\|) =$$
$$= \max(\|X_1'\| + \|X_2\|, \|X_1'\| + \|X_2'\|, \|X_1\| + 1 + \|X_2'\|) \le$$
$$\le \max(\|X\|, \|X_1'\| + \|X_2'\|, \|X_1\| + 1 + \|X_2'\|) = \|X\|.$$

Таким образом, восстановление зависимостей в функции `Final_Recover_Dependence` не увеличивает высоту графа зависимостей. Следовательно, приведённый алгоритм даёт минимально возможную высоту графа зависимостей для данных преобразований. Теорема доказана.

## 5. Сравнение с существующими алгоритмами

В этой главе разработанный алгоритм минимизации высоты графа зависимостей сравнивается с алгоритмами, описанными в главе 3.

Алгоритмы 1 и 1' обеспечивают минимальную высоту графа зависимостей и при этом строят минимальное число новых узлов. Однако они очень трудоёмки: первый имеет экспоненциальную сложность по числу возможных разрывов, второй же сводит задачу к классу задач целочисленного линейного программирования, которые являются NP-полными (по числу возможных разрывов). По этой причине данные алгоритмы малоприменимы. На практике их можно использовать только для небольших линейных участков, имеющих большое число повторений.

Далее оценим сложность Алгоритмов 2 и 3, начав с последнего.

Пусть $e$-количество дуг в графе зависимостей, $m$ - количество зависимостей, которые можно разорвать.

Первый шаг Алгоритма 3 (функция Break_Dependence) выполняется за $O(m)$ действий.

Второй шаг ( Mark_Early_Time_With_Partial_Recover_Dependence) требует обойти все дуги (обойти все узлы и от каждого узла обойти все выходящие из него дуги) – соответственно, его сложность *O(e)*. Кроме того, необходимо добавить $O(m)$ на анализ узлов, к которым шла разорванная зависимость. В результате имеем сложность $O(e+m)$.

На третьем шаге (Mark_Late_Time) также необходимо обойти все дуги (сложность $O(e)$).

На четвёртом шаге (Final_Recover_Dependence) для $m$ разорванных зависимостей в худшем случае может понадобиться обойти все узлы, и для каждого узла обойти всех последователей, то есть обойти все дуги, плюс ещё некоторое константное время на добавления, удаления из списков. При этом сложность $O(me \cdot const) = O(me)$. Однако в среднем коррекция будет быстро затухать и реально понадобится обойти лишь $r(e)$ дуг, где величина $r(e)$ существенно меньше $e$. Значение величины $r(e)/e$ будет оценено в 7 главе по результатам эксперимента. Здесь следует также заметить, что термин "среднее время" мы используем как интуитивное умозаключение, подтверждённое результатами эксперимента. Итак, суммарное время в среднем равно $O(m) \cdot r(e)$.

В худшем случае сложность Алгоритма 3 равна

$$O(m) + O(e+m) + O(e) + O(me),$$

а в среднем

$$O(m) + O(e+m) + O(e) + O(m) \cdot r(e) \tag{1}$$

Теперь оценим сложность Алгоритма 2. Разметка времени раннего и позднего планирования занимает $O(e)$. Затем в худшем случае необходимо $m$ раз найти фронт и $m$ раз произвести разметку времён после разрыва фронта. Всего в худшем случае на нахождение фронта понадобится $O(e)$ действий. Однако в среднем, при оптимистической оценке, подобной анализу предыдущего алгоритма, понадобится $r(e)$ действий. Итого, в худшем случае имеем

$$O(e) + O(em) + O(em) + O(m)$$

а в среднем $$O(e) + O(em) + O(m) \cdot r(e) + O(m)$$

Таким образом, в худшем случае Алгоритм 2 и Алгоритм 3 имеют одинаковую по порядку сложность, но в среднем Алгоритм 3 лучше.

## 6. Некоторые обобщения

В этой главе будут рассмотрены некоторые обобщения описанной техники разрыва зависимостей.

Выше рассматривались линейные участки с одним выходом, однако данная техника может без каких-либо изменений применяться к линейным участкам с произвольным числом выходов. При

этом будет сведена к минимуму высота всех выходов. Действительно, если в доказательстве приведенной выше теоремы множества $X_2$ и $X_2'$ переопределить как множества, ведущие не к END, а к произвольному выходу, то доказательство в основном не изменится. Новым в нем будет тот факт, что $lateA$ может не оказаться равным $\|X\|-\|X_2\|$, однако верно и то, что $\|X\|-\|X_2\| \geq lateA$, а потому всё равно будет выполняться неравенство $\|X\|-\|X_2\| \geq \|X_1'\|$, которое использовалось при доказательстве.

В приведённом способе разрыва зависимостей предполагалось, что длина дуги от операции, к которой идёт разрываемая зависимость, к вновь построенной операции равняется единице. Однако это не всегда выполняется. Например, пересылка (MOV) вещественного значения может длиться несколько тактов. В этом случае при разрыве зависимости может возникнуть замедление некоторых выходов. Это происходит, если $maxearly < early_p < maxearly + delay(A, A')$. Пусть из $A$ достижим только один выход – $BRANCH_1$, а из $A'$ достижим только другой выход – $BRANCH_2$. В этом случае решения о разрыве зависимости можно принимать на основе информации о вероятностях этих выходов. Разрыв зависимости необходимо оставить, если

$$prob_2 \cdot (maxearly + delay(A, A') - early_p) < prob_1 \cdot (early_p - maxearly),$$

где $prob_1$ - вероятность $BRANCH_1$, а $prob_2$ - вероятность $BRANCH_2$. Данное неравенство фактически означает, что разрыв этой зависимости поднял одну ветку выше, чем опустил другую. Пусть теперь из $A$ и $A'$ достижимы некоторые множества, которые, вероятно, пересекаются. Тогда можно применять приведённую выше формулу, однако в ней $prob_1$ и $prob_2$ будут суммарной вероятностью всех выходов, достижимых из $A$ и из $A'$, соответственно.

Необходимо заметить, что приведённая формула является эвристикой, то есть не обеспечивает минимальности графа зависимостей. Но, с другой стороны, её достоинство состоит в том, что решение о восстановлении зависимостей применяется локально и не требует обхода всего графа зависимостей.

## 7. Результаты эксперимента

Описанный Алгоритм 3 был реализован в двоичном оптимизирующем компиляторе для архитектуры "Эльбрус". Ниже приведены результаты анализа его работы на пакете тестов **Specperf**, содержащем наиболее значимые фрагменты пакетов тестов **Spec92**, **Spec95** и **Spec2000** 0. Исследовалась скорость работы алгоритма, являющаяся одной из его определяющих характеристик.

Было изучено поведение величины $r(e)$ в формуле (1). На Рис.3 приведён график распределения отношения этой величины к количеству дуг в графе, то есть функция распределения $r(e)/e$.

Как видно из графика, эта величина действительно достаточно мала, в среднем её значение составляет 0,02846. Таким образом, быстрый алгоритм, действительно, в среднем заметно лучше, чем Алгоритм 2.

Интересной также представляется зависимость отношения $r(e)/e$ от числа дуг в графе зависимостей. График отображения $e \mapsto r(e)/e$ на наших статистических данных представлен на Рис.4.

Рис.4 показывает, что при использовании разработанного алгоритма не происходит роста величины $r(e)/e$ с увеличением числа дуг в графе зависимостей, то есть и при больших $e$ в среднем Алгоритм 3 быстрее Алгоритма 2.

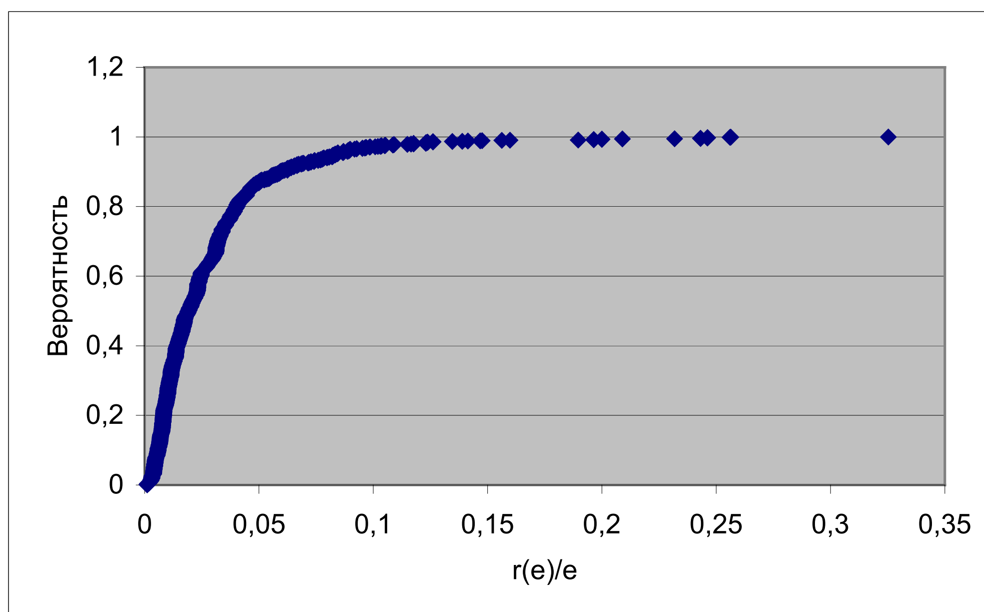



Рис.3. Функция распределения величины $r(e)/e$

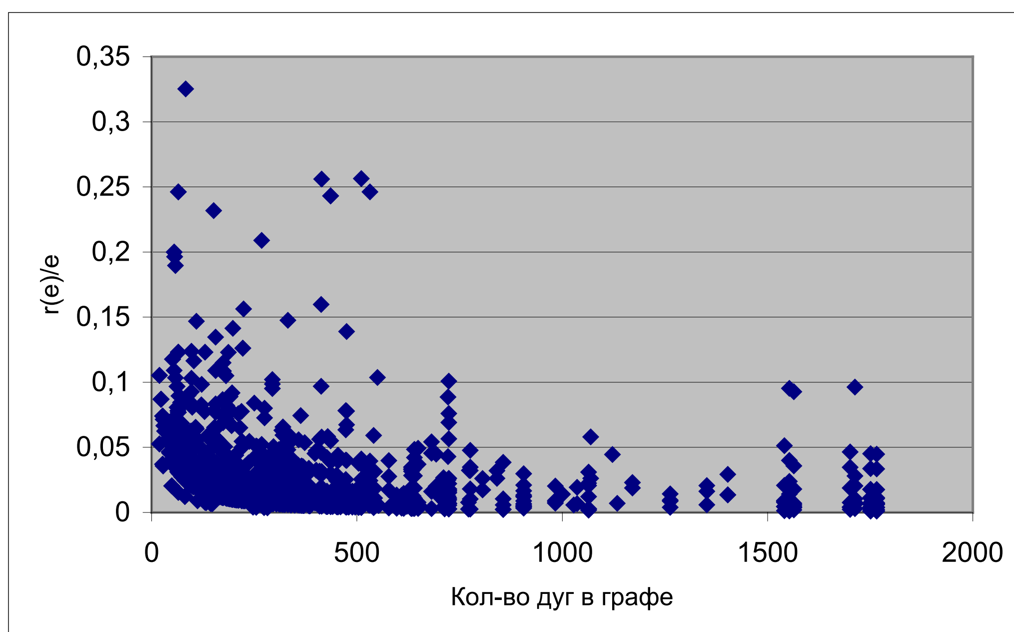



Рис.4. Зависимость величины $r(e)/e$ от числа дуг в графе зависимостей

## Заключение

В работе исследована задача минимизации высоты графа зависимостей с помощью разрыва зависимостей путём построения новых операций. Произведена формализация проблемы на языке графов. Приведён новый быстрый алгоритм минимизации высоты графа зависимостей и доказана его корректность. Описаны альтернативные алгоритмы для решения данной задачи. Произведены оценки сложности всех описанных алгоритмов в худшем случае, а также оценки сложности двух алгоритмов в среднем. Проведён сравнительный анализ двух последних оценок. В эксперимен-

тальной части исследован выигрыш в скорости работы алгоритма, который получается в результате использования быстрого алгоритма.

Из результатов эксперимента следует, что новый быстрый алгоритм минимизации высоты графа зависимостей действительно асимптотически заметно быстрее своих аналогов. Другой важной особенностью алгоритма является его универсальность - его можно применять для разрыва любых зависимостей, удовлетворяющих некоторым формальным условиям (они описаны в главе 2). Например, как уже говорилось, алгоритм применим для реализации оптимизации unzipping 0.

## Литература


1. SebastianWinkel. "Optimal Global Scheduling for ItaniumTM Processor Family". Second Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology, 2002.
2. Daniel Kästner and Sebastian Winkel. ″ILP-based Instruction Scheduling for IA-64". Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embeded Systems, 2001.
3. Rakesh Ghiya, Daniel Lavery, and David Sehr. "On the Importance of Point-to Analysis and Other Memory Disambiguation Methods for C Programs". Proceedings of the ACM SIGPLAN ´01 Conference on Programming Language Desing and Implementation (PLDI), pages 47-58, June 2001.
4. Гимпельсон В.Д. , Масленников Д.М. ″Быстрый алгоритм минимизации высоты графа зависимостей″. Сборник тезисов XXI научно-технической конференции войсковой части 03425. Москва, в/ч 03425, 2003г.
5. Масленников Д.М., Василец П.С., Гимпельсон В.Д., Матвеев П.Г., Муслинов Р.Г. ″Программно-аппаратный метод обеспечения точного состояния контекста при прерываниях в двоично-оптимизированном коде″. Сборник тезисов XXI научно-технической конференции войсковой части 03425. Москва, в/ч 03425, 2003г.
6. Keith Diefendorff. "The Russians Are Coming". Microprocessor Report 15.02.1999.
7. Intel. ″Intel® Itanium® Architecture Software Developer's Manual". Oct. 2003.
8. Intel. "Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization". Apr. 2003.
9. Baraz L. et al, "IA-32 Execution Layer: a Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems". Proceedings of the 36th International Symposium on Microarchitecture, 2003.
10. Marks, M. et al, "Binary Translation". Digital Technical Journal, Vol. 4, No. 4, Special Issue, 1992, pp. 1-24.
11. Hookway, R., and Herdeg, M., "DIGITAL FX!32: Combining Emulation and Binary Translation". Digital Technical Journal, Vol. 9, No. 1, August 28, 1997, pp. 3-12.
12. Chernoff, A. et al, "FX!32 A Profile-directed Binary Translator". IEEE Micro, March/April 1998, pp. 56-64.
13. Paul J. Drongowski, David Hunter, Morteza Fayyazi, David Kaeli. "Studying the Performance of the FX!32 Binary Translation System". Proceeding of the 1st Workshop on Binary Translation, Oct. 1999.
14. Eric R. Altman, Kermal Ebcioglu, Michael Gschwind and Sumedh Sathaye. "Advances and Future Challenges in Binary Translation and Optimization". Proceeding of the IEEE Special Issue on Microprocessor Architecture and Compiler Technology, November 2001.
15. Klaiber, A., "The Technology Behind Crusoe Processors". Transmeta Corporation white paper, January 2000
16. Dehnert J.C., Grant B.K., Banning J.P., Johnson R., Kistler T., Klaiber A, and Mattson J. "The transmeta code morphing software: using speculation, recovery and adaptive retranslation to address real-life challenges". Proceedings of the International Symposium on Code Generation and Optimization, 2003.
17. Michael Schlansker and Vinod Kathail. ″Critical Path Reduction for Scalar Programs″. Proceedings of the 28th International Symposium on Microarchitecture. November, 1995.
18. David I. August, Wen-mei W. Hwu, and Scott A. Mahlke. ″A Framework for Balancing Control Flow and Predication″. Proceedings of the 30th International Symposium on Microarchitecture. December, 1997.
19. Волконский В.Ю, Окунев С.К. ″Оптимизация критического пути на предикатном представлении программы″. Информационные технологии, № 9. Москва, сентябрь 2003.
20. Muchnick S.S. "Advanced compiler desing and implementation". Morgan Kaufmann Publishers, 1997.
21. Касьянов В.Н., Евстигнеев В.А. "Графы в программировании: обработка, визуализация и применение". СПб.: БХВ-Петербург, 2003.
22. http://www.specbench.org/.



**Гимпельсон Вадим Дмитриевич.** Родился в 1981 году. В 2003 году окончил МГУ им. М.В. Ломоносова. Автор 2 научных работ. Аспирант ИМВС РАН.

**Масленников Дмитрий Михайлович.** Родился в 1967 году. В 1992 году окончил МИЭМ. Автор 7 научных работ. Заведующий отделением ИМВС РАН