МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Государственное образовательное учреждение
высшего профессионального образования
«Ростовский государственный университет»

Механико-математический факультет
Кафедра алгебры и дискретной математики

Магистерская диссертация

Автоматическое разрезание и слияние программных циклов

Студента 2 года магистратуры	Бутов А. Э.
Научный руководитель к.ф.-м.н., доц. каф. АДМ мехмата РГУ	Штейнберг Б. Я.
Рецензенты к.ф.-м.н., доц. каф. ИВЭ мехмата РГУ	Шевченко И. В.
к.ф.-м.н., асс. каф. АДМ мехмата РГУ	Максименко Е. А.

Ростов-на-Дону
2005 г.

Содержание:

Введение. 3

1 Разработка сервисных функций Открытой распараллеливающей системы. . 8

2 Разрезание программных циклов 9

2.1 Примеры применения разрезания циклов для разных вычислительных архитектур 10

2.2 Условия корректности и эквивалентности преобразования 12

2.3 Алгоритм преобразования. 14

2.4 Использование вспомогательных преобразований для разрезания 24

2.4.1 Растягивание скаляров 24

2.4.2 Расщепление вершин (Введение временных массивов) 30

2.4.3 Применение динамического программирования 32

2.5. Вынос инварианта из цикла 33

3 Слияние программных циклов 37

3.1 Область использования слияния 38

3.2 Условия корректности и эквивалентности преобразования 39

3.3 Алгоритм преобразования 39

3.4 Случаи сложных преобразований 42

4 Выделение векторизуемых циклов 46

4.1 Примеры векторизуемых циклов 46

4.2 Функция нахождения векторизуемых циклов 47

5 Исследование разрезаемости случайных циклов 48

5.1 Генерация случайных циклов 49

5.2 Результаты исследования разрезаемости случайных циклов 51

Заключение 55

Литература 56

Введение.

В данной работе рассмотрены распараллеливающие преобразования разрезание и слияние программных циклов, а также ряд вспомогательных преобразований. Получена их программная реализация во внутреннем представлении Открытой распараллеливающей системы (ОРС). Реализованный алгоритм разрезания программных циклов включает в себя автоматическую перестановку операторов, способствующую разрезанию. Написано вспомогательное преобразование «растягивание скаляров», которое позволяет некоторые неразрезаемые циклы привести к разрезаемому виду. В рамках данной работы программно реализованы сервисные функции, которые предназначены для выполнения таких операций, как поиск, вставка, копирование, удаление и замена фрагментов внутреннего представления. Эти операции используются в разрезании, слиянии и преобразованиях программ других разработчиков ОРС. На основе разрезания написано преобразование «вынос инварианта из цикла» и функция выделения векторизуемых циклов. Написана программа, строящая во внутреннем представлении ОРС случайные циклы (с заданными параметрами), и с помощью нее проведено исследование частоты разрезаемости случайных циклов, а также эффективности вспомогательных преобразований. Для тестирования преобразований ОРС, в рамках данной работы, написано преобразование кода во внутреннем представлении в язык С (депарсер).

Суть разрезания программных циклов заключается в замене фрагмента программы, состоящего из цикла, в теле которого много операторов присваивания, преобразованным фрагментом, состоящим из нескольких циклов, тела которых содержат меньше операторов присваивания [7]. Это одно из наиболее важных преобразований, которое может быть использовано при распараллеливании программ на практически все типы параллельных компьютеров: векторные, конвейерные, MIMD, SIMD и т.д. Если исходный

цикл был не параллельный, то есть шанс, что в результате данного преобразования некоторые из полученных циклов будут параллельны. Например, для некоторых конвейерных машин неконвейеризуемые циклы, в теле которых слишком много операций, с помощью разрезания могут быть приведены к нескольким конвейеризуемым. Не всегда преобразование разрезания можно применить непосредственно. В этом случае к исходному циклу предварительно применяются вспомогательные преобразования: перестановка операторов, растягивание скаляров, расщепление вершин и т.д. Перестановка операторов входит в реализованный алгоритм разрезания. Растягивание скаляров заключается в замене скаляров внутри цикла временными массивами. Таким образом, можно избавиться от некоторых зависимостей. Была написана функция, разрезающая цикл на мельчайшие части. Если, после разрезания, какой либо из полученных циклов содержит более одного оператора, функция применяет к нему по очереди вспомогательные преобразования и рекурсивно вызывает сама себя.

Преобразование «вынос инварианта из цикла» [16] является оптимизирующим. Оно использует разрезание для отделения инвариантов из тела цикла (если операторов в теле цикла больше одного).

Слияние программных циклов является преобразованием обратным к разрезанию [12], [13], [14]. При этом фрагмент программы, состоящий из двух циклов, заменяется фрагментом из одного цикла, содержащего операторы первого и второго циклов исходного фрагмента.

Было написано также вспомогательное преобразование перестановки фрагментов программы. Суть перестановки состоит в том, чтобы, если это возможно, поменять местами два последовательно идущих фрагмента программы с сохранением семантической корректности (эквивалентности исходной программы полученной). В связи с тем, что данное преобразование не всегда корректно, для отслеживания эквивалентности используется граф информационных зависимостей.

Возможны также более сложные комбинированные преобразования, использующие разрезание и слияние, отщепление итераций, перестановку программных фрагментов и другие преобразования из библиотеки преобразований ОРС.

ОРС разрабатывается студентами и аспирантами кафедры алгебры и дискретной математики. Эта система предназначена для автоматического распараллеливания программ с процедурных языков программирования (ФОРТРАН, Паскаль, Си) на параллельные суперкомпьютеры. Автоматически распараллеливающие и оптимизирующие преобразования являются центральным этапом обработки исходного программного кода в ОРС.

ОРС состоит из нескольких частей: парсера (разборщика), библиотеки преобразований и генератора кода. Парсер, используя грамматику входного языка, преобразует исходный текст программы во внутреннее представление. Во внутреннем представлении программа выглядит как совокупность древовидных структур, узлы которой представляют операторы, операции и идентификаторы языка, а дуги между ними представляют отношения принадлежности и вложенности. Для внутреннего представления характерно то, что его легко можно перевести обратно в исходный текст, а также над ним легко совершать такие редакторские операции как удаление, вставка, копирование и замена.

Библиотека преобразований - это группа модулей, каждый из которых представляет собой некоторое преобразование фрагмента программы во внутреннем представлении. Преобразования, используя сервисные функции (простейшие редакторские операции над внутренним представлением), изменяют программу, либо фрагмент программы. Основная задача преобразований - улучшение некоторых качеств программы: либо повышение быстродействия, либо уменьшение объема требуемой памяти, либо повышение точности вычислений, либо получение исходного текста на

другом языке программирования (для последующего использования в какой-то программной среде).

Так как внутреннее представление имеет структуру дерева, то и сервисные функции представляют собой функции работы с деревом (обход дерева, вставка, копирование, удаление, замена ветвей). Однако, в связи с общим контекстом данных преобразований, как преобразований фрагментов программы, сервисные преобразования разделены на две части: преобразования выражений (копирование, вставка, поиск подвыражений и т.п.) и преобразования операторов (копирование, удаление операторов, подсчет количества операторов в блоке и т.п.).

Преобразования, которые улучшают быстродействие программы на обычных последовательных машинах, называются оптимизирующими [5], [6]. К таким преобразованиям относится, например, протягивание констант.

Преобразования, задачей которых является выявление и, если это возможно, улучшение параллелизма программы [1], [2], [3] (то есть увеличение эффективности её работы на суперкомпьютере), называются распараллеливающими. Так как существует множество различных семейств суперкомпьютерных платформ, то и распараллеливающие преобразования бывают специфичными для какого-то одного семейства. В связи с чем, некоторые преобразования выполняют прямо противоположную работу. Например, преобразование разрезания цикла и преобразование слияния циклов являются взаимообратными. Поэтому целесообразность применения того или иного преобразования может зависеть от целевой платформы. Некоторые преобразования могут использоваться как для распараллеливания, так и для оптимизации, например, раскрутка цикла.

Все преобразования работают с внутренним представлением, поэтому нет необходимости отслеживать синтаксическую корректность преобразованных фрагментов программы, но в то же время каждое преобразование должно проверять эквивалентность преобразованных фрагментов исходным. Для решения этой задачи используется анализ

информационных зависимостей в программе. Для описания зависимостей используются графы информационных связей (граф Лампорта, решетчатый граф) и граф выполнения программы. Генераторы кода (депарсеры) также работают с внутренним представлением, однако их нельзя отнести к преобразованиям, так как на выходе у них не внутреннее представление, а программный код.

Автоматическое распараллеливание позволяет:

- ускорить время разработки параллельных программ;

- понизить требования к квалификации программистов, пишущих параллельные программы, поскольку от них можно скрыть многие особенности конкретной параллельной архитектуры;

- повысить надежность разрабатываемых программ, т.к. объем исходного текста для последовательной программы на порядок меньше, чем параллельной, и логическая структура ее проще;

- перенести некоторые разработанные программы с последовательных компьютеров на параллельные.

1 Разработка сервисных функций Открытой распараллеливающей системы.

Все преобразования программ в ОРС осуществляются во внутреннем представлении. Оно представляет собой совокупность деревьев. Это дерево операторов и дерево идентификаторов. Узлами дерева операторов являются операторы и операции входного языка. Узлами дерева идентификаторов – области видимости и идентификаторы. При этом оба дерева тесно связаны.

Для удобной работы с внутренним представлением были написаны сервисные функции, выполняющие простейшие редакторские операции. По сути, это функции работы с деревом, такие как обход дерева с подсчетом узлов, удаление, копирование, замена поддеревьев и т.п. Однако то, что деревья являются на самом деле способом представления программы во внутреннем представлении, накладывает свои требования на интерфейс и содержание данных функций. Все сервисные функции логически разделены на две независимые части: функции работы с выражениями и функции работы с операторами. При этом для второго блока функций необходимо вносить изменения не только в граф операторов, но и в граф идентификаторов.

Пример 1

Например, при копировании следующего цикла

```
for (i=0; i<100; i++)
{
  int c=20;
  A[i]=B[c+i];
}
```

необходимо учитывать, что в его теле объявляется переменная ‘c’. Поэтому необходимо скопировать не только сам цикл, но и создать в дереве

идентификаторов область видимости, связанную с телом данного цикла, добавить туда идентификатор ‘c’ и связать с ним все обращения к переменной ‘c’ в скопированном цикле.

Были написаны следующие сервисные функции для работы с выражениями:

1) сравнения выражений;

2) копирования выражения;

3) замены одного выражения другим;

4) поиск подвыражений;

5) автоматического поиска и замены выражений (находит и заменяет между двумя операторами все вхождения одного выражения на другое),

а также сервисные функции для работы с операторами:

1) удаления оператора;

2) копирования оператора перед, либо после указанного оператора;

3) замены оператора (заменяет один оператор другим);

4) добавления оператора в начало или конец блока операторов;

5) нахождения количества операторов в блоке.

2 Разрезание программных циклов

Для большинства архитектур наибольшее время работы программы отнимают циклически повторяющиеся участки. Поэтому в оптимизирующих и распараллеливающих компиляторах наиболее распространены преобразования циклов. Разрезание цикла - одно из наиболее важных преобразований, которое может быть использовано при распараллеливании программ на практически все типы параллельных компьютеров: векторные, конвейерные, MIMD, SIMD и т.д.

Суть разрезания цикла состоит в том, чтобы заменить цикл, в теле которого много операторов присваивания, на эквивалентный фрагмент программы из нескольких циклов, в телах которых меньше операторов [7].

При распараллеливании зачастую большой цикл не может быть эффективно отображен на архитектуру суперкомпьютера (например, из-за нехватки ресурсов). В этом случае есть надежда, что после разрезания хотя бы некоторые из результирующих циклов смогут быть параллельно вычислены.

В рамках данной работы была получена программная реализация данного преобразования. Она входит в пакет преобразований ОРС отдельным модулем LoopSplitting.

Не всякий цикл можно разрезать непосредственно. Иногда для этого следует применить вспомогательные преобразования [1, с. 66-67]. Разрезание цикла позволяет получить и частичную векторизацию цикла. Разрезание кратных циклов очевидным образом сводиться к разрезанию одномерных. Иногда вместо термина «разрезание» используется термин «разбиение» («fission» [12], «distribution» [14]) .

Функция разрезания цикла пытается разрезать цикл на максимальное число частей, затем к каждому из полученных циклов, в теле которых осталось более одного оператора присваивания, по очереди применяются вспомогательные преобразования, а затем функция снова пробует их разрезать.

2.1 Примеры применения разрезания циклов для разных вычислительных архитектур

2.1.1 Разрезание циклов применяется при распараллеливании программ на практически все типы параллельных компьютеров.

Для векторных суперкомпьютеров разрезание применяется для отделения векторизуемых циклов (более подробно о векторизации циклов рассматривается в разделе 5).

Пример 1

```
for (i=0; i<=9; i=i+1)
{
  A[i]=B[i]+1;
  B[i]=B[i-1]+i;
}
```

Второй оператор рекурсивный, поэтому данный цикл не может быть выполнен на векторной машине эффективно. Применив к этому циклу преобразование разрезания, получим два цикла:

```
for (i=0; i<=9; i=i+1)
  A[i]=B[i]+1;
for (i=0; i<=9; i=i+1)
  B[i]=B[i-1]+i;
```

первый из которых может быть выполнен одной векторной командой.

2.1.2 Точно также разрезание используется при распараллеливании на архитектуры SIMD и MIMD. При этом отделяются циклы, итерации которых можно распределить по разным процессорам.

Пример 2

```
for (i=0; i<=9; i=i+1)
{
  A[i]=B[i]+A[i-2];
  B[i]=C[i+1]+k;
}
```

Итерации цикла нельзя разделить между процессорами из-за рекуррентности первого оператора. Разрезав данный цикл, получим два цикла:

```
for (i=0; i<=9; i=i+1)
  A[i]=B[i]+A[i-2];
for (i=0; i<=9; i=i+1)
  B[i]=C[i+1]+k;
```

Теперь итерации второго цикла можно распределить между процессорами.

2.1.3 Для однопроцессорных машин разрезание может быть использовано для оптимизации. В примере 2 после разрезания для второго цикла может применено оптимизирующее преобразование развертки (реализованное в ОРС Науменко Сергеем Алексеевичем), которое не могло быть применено к изначальному циклу.

2.1.4 Возможно применение разрезания при распараллеливании на конвейерные машины [11]. В случае, когда тело цикла содержит больше простых операций (+, -, * и т.п.), чем элементарных элементов конвейера, разрезание может помочь отделить циклы с необходимым количеством операций.

2.2 Условия корректности и эквивалентности преобразования

Разрезание цикла заменяет цикл

```
for (i=0; i<N; i++)
{
  S1;
  ..........
  Sk;
  S(k+1);
  ..........
  Sm;
}
```

на фрагмент программы, состоящий из последовательности двух циклов

```
for (i=0; i<N; i++)
{
  S1;
  ..........
  Sk;
}
for (i=0; i<N; i++)
{
  S(k+1);
  ..........
  Sm;
}
```

Условия применения:

1) каждый из фрагментов программы S1,...,Sk и S(k+1),...,Sm имеет один вход и один выход. В частности:

1.1. (для фортрана) ни один из операторов S(k+1),...,Sm не находится в зоне действия какого-либо условного оператора из множества S1,...,Sk. (т.е. выполнение или невыполнение этого оператора не зависит от значения логического выражения условного оператора)

1.2. (для фортрана) ни один из операторов S(k+1),...,Sm не находится в теле цикла, заголовок которого во множестве S1,...,Sk

1.3. всякий оператор goto из множества S1,...,Sk (или S(k+1), ..., Sm) указывает на метку оператора из этого же множества.

2) не существует такой дуги графа информационных связей (v1,v2), что v1 принадлежит Si, k+1=<i=<m, v2 принадлежит Sj, 1=<j=<k.

Условия применимости данного преобразования для программы во внутреннем представлении будут несколько иные.

2.3 Алгоритм преобразования.

В программе реализован следующий алгоритм преобразования разрезания:

1) проверяем, является ли входной цикл канонизированным (он должен быть канонизирован предварительно);
2) в зависимости от установленных при вызове процедуры флагов, применяем вспомогательные преобразования (растягивание скаляров, временные массивы);
3) строим фактор-граф по компонентам сильной связности графа информационный зависимостей;
   3.1. строим граф информационных зависимостей Лампорта по телу данного цикла;
   3.2. строим матрицу смежности по этому графу;
   3.3. используя алгоритм Флойда, строим матрицу достижимости;
   3.4. находим по ней компоненты сильной связности и формируем список;
   3.5. сортируем компоненты в списке в порядке направления дуг зависимости между ними (используя матрицу достижимости);
4) копируем исходный цикл столько раз, сколько компонент в полученном списке;
5) оставляем в каждом i-ом скопированном цикле только операторы из i-ой компоненты, остальные удаляем.

Пример 1

Разберем алгоритм более подробно на примере следующего фрагмента программы:

```
for (i=0; i<=9; i=i+1)
{
  A[i]=B[i]+1;
  B[i+1]=4;
  B[i]=A[i+1]+1;
}
```

1) Программный цикл типа for будем называть канонизированным, если его левая граница равна 0, правая граница – константа, а шаг равен 1. Черданцевым Денисом Владимировичем написана функция канонизации цикла. Она, наряду с другими предварительными преобразованиями, применяется сразу после компиляции программы во внутреннее представление, а также после некоторых преобразований, которые могут нарушить каноничность циклов.

2) Если при вызове процедуры разрезания были включены флаги использования вспомогательных преобразований растягивания скаляров (Var2Array) и временных массивов (TempArrays) (по умолчанию выключенные), то перед выполнением непосредственно разрезания, к данному фрагменту применяются соответствующие преобразования. Об этих вспомогательных преобразованиях речь будет идти в пункте 3.2.

3) В данной работе используется анализ информационных зависимостей, поэтому необходимо напомнить понятия информационной зависимости и графа информационных зависимостей теории оптимизирующих (распараллеливающих) преобразований [8], [9].

Вхождением переменной будем называть всякое появление переменной в тексте программы вместе с тем местом в программе, в котором эта переменная появилась. Всякому вхождению при конкретном значении

индексного выражения соответствует обращение к некоторой ячейке памяти. Если при этом обращении происходит запись в ячейку памяти (вхождение в левую часть оператора присваивания, не входящее в индексное выражение другого вхождения), то такое вхождение называется *генератором*. Остальные вхождения называются *использованиями*.

Пример 2

В следующем операторе присваивания

```
A[I + B[J + 2]] = D[I - 1, B[I]] + 3*A[1] – I + 5
1 2   3 4           5 6       7 8        9      10
```

десять вхождений переменных (их номера проставлены снизу) – и только первое является генератором.

Говорят, что *два вхождения порождают информационную зависимость*, если при некоторых допустимых значениях индексных выражений они обращаются к одной и той же ячейке памяти.

Пример 3

```
for (i=0; i<100; i++)
    A[i]=D[i, i-1]+A[i-1];
```

В данном операторе вхождения переменной `A` информационно зависимы, поскольку оба обращаются к ячейке памяти `A[10]`: вхождение `A[I]` на десятой итерации, а `A[I-1]` - на одиннадцатой итерации цикла.

Граф информационных связей – это ориентированный граф, вершины которого соответствуют вхождениям, а дуга соединяет пару вершин `(v,u)`, если выполняется одно из следующих условий:

- эти вхождения обращаются к одной и той же ячейке памяти (т.е. порождают информационную зависимость), причем вхождение `v` раньше, чем `u` и хотя бы одно из этих вхождений является генератором;

- вхождение u является генератором, а вхождение v принадлежит этому же оператору присваивания. Такие дуги будем называть тривиальными.

Генератор будем обозначать – out (output), а использование - in (input). Дуги графа информационных связей бывают трех типов в зависимости от типов инцидентных им вершин: out-in - *истинная информационная зависимость* (true dependence), in-out - *антизависимость* (antidependence), out-out - *выходная зависимость* (output dependence).

Если информационная зависимость связывает два использования, то мы будем говорить об in-in зависимости.

Иногда бывает трудно определить, существует или нет информационная зависимость между парой вхождений. При преобразованиях программ в таких случаях предполагают худшее – считают, что такая зависимость существует, и на графе соответствующие вершины соединяют дугой.

Пример 4

```
for (i=0; i<100; i++)
    C[2*i]=C[2*i+k];
```

В этом примере если k - нечетное, то между обоими вхождениями переменной X нет информационной зависимости; если k - четное и отрицательное, то есть истинная информационная зависимость, делающая цикл рекуррентным; если k - четное и неотрицательное, то имеет место антизависимость. Если до выполнения программы неизвестно, какое значение примет k к началу выполнения цикла, то и характер зависимости тоже нельзя определить. В этом случае в графе информационных зависимостей должны быть проведены обе дуги между двумя вхождениями

переменной X. Также, для таких случаев применяется динамическое программирование.

Пример 5

В следующем цикле дуга графа информационных связей ведет от первого вхождения A[i] ко второму, но не наоборот. А вхождения переменной X соединяются двумя дугами в обе стороны: дуга антизависимости, поскольку на каждой итерации сначала X используется в первом операторе тела цикла, а потом перезаписывается; дуга истинной зависимости, поскольку на каждой итерации, начиная со второй, используется значение X, полученное на предыдущей итерации.

```
for (i=0; i<N; i++)
{
    A[i]=B+X;
    X=A[i];
}
```

Информационная зависимость между вхождениями называется *циклически независимой* (loop independent dependence), если эти вхождения обращаются к одной и той же ячейке памяти на одной и той же итерации цикла. Иначе зависимость называется *циклически порожденной* (loop carried dependence).

В предыдущем примере дуга зависимости между вхождениями A[i] циклически независима, дуга зависимости между вхождениями X, ведущая сверху вниз тоже циклически независима, а дуга, ведущая снизу вверх – циклически зависима.

Пусть v1 и v2 – два вхождения массива X в тело цикла со счетчиком i, связанные информационной зависимостью. Информационную зависимость между v1 и v2 будем называть *регулярной* относительно этого цикла, если разность индексных выражений вхождений v1 и v2 не зависит от счетчика цикла i.

Для анализа информационных зависимостей в программе используется граф Лампорта, реализованный Шульженко Александром в рамках проекта ОРС.

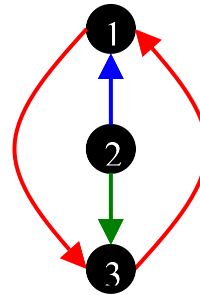
Построение фактор-графа проводится в следующей последовательности:

3.1. Строим граф информационных зависимостей Лампорта для рассматриваемого цикла. Изобразим графически информационные зависимости для данного фрагмента:

```
for (i=0; i<=9; i=i+1)
{
  A[i] = B[i] + 1;

  B[i+1] = 4;

  B[i] = A[i+1] + 1;
}
```



Потоковая зависимость показана синим цветом, выходная - зеленым , антизависимости - красным. Из данного рисунка видно, что первый и последний операторы не могут быть разнесены по разным циклам. Второй оператор должен выполняться до первого и последнего и может быть вынесен в отдельный цикл. Если представить тело цикла в виде графа с вершинами операторами и дугами – зависимостями между вхождениями, входящими в эти операторы, то получим граф, изображенный на рисунке справа.

3.2. Используя функции нахождения зависимости по двум операторам, проходим по всем операторам тела цикла и строим матрицу смежности графа информационных зависимостей. В i-ой строке j-ом столбце матрицы записываем 1 (true), если в графе информационных зависимостей существует

дуга с началом во вхождении, принадлежащем i-ому оператору, и концом во вхождении, принадлежащем j-ому, иначе 0 (false).

Построим матрицу смежности для данного примера:

$$A = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

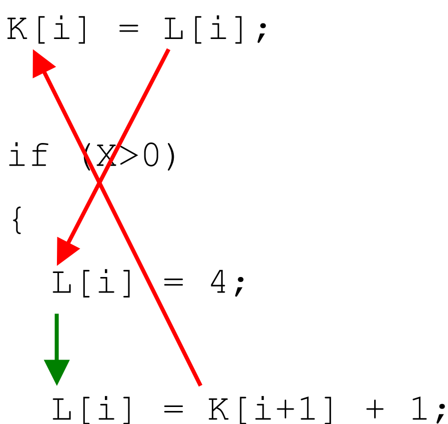
Если оператор тела цикла представляет собой оператор условия, цикла или список операторов, то при построении матрицы смежности он представляется как один оператор. При этом все операторы присваивания данного сложного оператора стягиваются в один узел, в котором начинаются и заканчиваются все дуги, которые ранее принадлежали простым операторам, его составляющим.

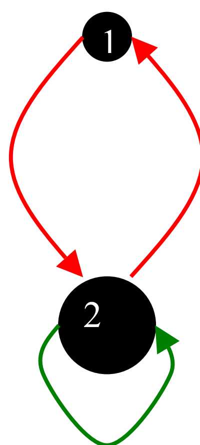
Пример 6 **Граф** **Матрица смежности**

```
for (i=0; i<=9; i=i+1)
{
  K[i] = L[i];

  if (X>0)
  {
    L[i] = 4;

    L[i] = K[i+1] + 1;
  }
}
```





$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

3.3. Строим матрицу достижимости, по графу Лампорта. В i-ой строке j-ом столбце матрицы достижимости записываем 1 (true), если в графе

информационных зависимостей существует путь с началом во вхождении, принадлежащем i-ому оператору, и концом во вхождении, принадлежащем j-ому, иначе 0 (false). Для построения матрицы достижимости по матрице смежности используется видоизмененный алгоритм Флойда (вместо операций сложения и умножения используются бинарные операции конъюнкции и дизъюнкции).

Для рассматриваемого примера получаем:

$$B = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

3.4. Две вершины графа i и j принадлежат одной и той же компоненте сильной связности, если существуют пути из первой вершины во вторую и из второй в первую. Т.е. необходимо, чтобы в матрице достижимости этого графа, были единицы в ячейках b(i, j) и b(j, i). Применив конъюнкцию поэлементно к элементам матрицы выше главной диагонали и соответствующим элементам ниже ее, получим в результате верхнетреугольную матрицу K (где k(i, j)=b(i, j) * b(j, i), для $i < j$ и k(i, j)=0, для $i \geq j$).

Для этого примера:

$$K = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

По этой матрице получается, что два оператора i и j ($i<j$) принадлежат одной компоненте сильной связности, если k(i, j)=1. Строится вектор, элементы которого соответствуют операторам. В первый элемент вектора записывается 1, также 1 записывается во все элементы вектора, соответствующие операторам, находящимся в одной компоненте сильной

связности с первым. Затем находится первый элемент вектора, в котором еще не записана 1. Он и все элементы, лежащие с ним в одной компоненте, помечаются 2-ой, и т.д. В итоге получается вектор, в котором каждая компонента сильной связности помечена своим числом.

Для данного примера вектор компонент имеет вид:

c=(1, 2, 1),

т.е. первый и третий оператор принадлежат первой компоненте, а второй оператор – второй компоненте.

3.5. Для расстановки компонент сильной связности в порядке следования используется матрица достижимости B. Если на графе информационных зависимостей существует дуга, выходящая из оператора, принадлежащего i-ой компоненте, и заканчивающаяся в операторе, принадлежащем j-ой компоненте, то для любого оператора i-ой компоненты существует путь, ведущий из нее в любой оператор j-ой компоненты. Пусть i_k оператор, принадлежащий i-ой компоненте, j_l оператор, принадлежащий j-ой компоненте, тогда в матрице смежности $b(i_k, j_l)=1$. Используя это свойство, расставляем компоненты в векторе в порядке направления дуг между ними, т.е. так, чтобы $b(i_k, j_l)=0$ (i_k - операторы i-ой компоненты, j_l - операторы j-ой компоненты) для всех $i>j$.

Упорядоченный вектор компонент для рассматриваемого примера строится следующим образом:

По матрице $B=\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$ находится, что b(2, 1)=1, причем 2-ой оператор принадлежит 2-ой компоненте, а 1-ый оператор – 1-ой, поэтому компоненты 1 и 2 меняются местами, в результате получаем упорядоченный вектор компонент v=(2, 1, 2).

4) Если количество компонент равно n, то исходный цикл копируется сам после себя n-1 раз.

В рассматриваемом случае n=2, значит цикл копируется 1 раз и получается следующий фрагмент:

```
for (i=0; i<=9; i=i+1)
{
  A[i]=B[i]+1;
  B[i+1]=4;
  B[i]=A[i+1]+1;
}
```

```
for (i=0; i<=9; i=i+1)
{
  A[i]=B[i]+1;
  B[i+1]=4;
  B[i]=A[i+1]+1;
}
```

5) В i-ом скопированном цикле оставляем только операторы, принадлежащие i-ой компоненте.

Результатом работы алгоритма для данного цикла будет следующий фрагмент программы:

```
for (i=0; i<=9; i=i+1)
{
  B[i+1]=4;
}
```

```
for (i=0; i<=9; i=i+1)
{
  A[i]=B[i]+1;
  B[i]=A[i+1]+1;
}
```

2.4 Использование вспомогательных преобразований для разрезания

Иногда разрезание цикла нельзя применить непосредственно, этому мешают информационные зависимости, которые можно удалить, используя специальные вспомогательные преобразования. К ним относятся преобразования «увеличение размерности массивов» (в частном случае «растягивание скаляров»), «временные массивы» («расщепление вершин»). Также для удаления информационных зависимостей применяются принципы динамического программирования.

2.4.1 Растягивание скаляров

В рамках данной работы было реализовано преобразование растягивание скаляров. Программная реализация, в виде отдельного модуля Var2Array входит в пакет преобразований ОРС. Суть данного преобразования состоит в замене скалярных переменных в теле цикла массивами, зависящими от счетчика цикла. Данное преобразование полезно для разрезания, так как удаляет некоторые дуги зависимости.

Алгоритм преобразования:

1) проверяем, является ли входной цикл канонизированным;

2) проходим по телу цикла и сохраняем информацию обо всех скалярных переменных, встречающихся в теле цикла вместе с полной информацией об их вхождениях;

3) для каждой найденной скалярной переменной:

3.1. проверяем, является ли переменная счетчиком данного цикла. Если да, то переходим к следующей переменной;

3.2. проверяем, есть ли вхождения данной переменной в левые части операторов присваивания тела данного цикла. Если нет, то делать данное преобразование не нужно, так как скалярная переменная не создает «вредоносных» информационных зависимостей;

3.3. создаем новый массив с уникальным именем (начинающимся с имени этой скалярной переменной). Размер массива равен правой границе цикла плюс 1;

3.4. инициализируем первый элемент массива значением скалярной переменной перед телом цикла;

3.5. в теле цикла заменяем все вхождения скалярной переменной на элементы массива;

3.6. добавляем после цикла оператор присваивания, в котором присваиваем скалярной переменной значение последнего элемента массива.

Разберем алгоритм преобразования более подробно на следующем примере:

Пример 1

```
for (i=0;i<10;i++)
{
  A[i]=B[i]+C;
  C=A[i-1];
  B[i]=A[i]+C;
}
```

В данном цикле помимо счетчика цикла есть скалярная переменная C, которая мешает разрезанию.

1) Проверяем, является ли входной цикл канонизированным.

2) Проходим по телу цикла (обход дерева в глубину) и записываем в ассоциативный массив (структура map) переменные вместе со списком их вхождений. Для каждого вхождения записывается указатель на это вхождение, указатель на содержащий его оператор присваивания и номер вхождения в данном присваивании (если номер вхождения равен нулю, то данное вхождение генератор, иначе - использование).

Для рассматриваемого цикла будет сохранена примерно такая информация:
i: (первый оператор, вхождение №1), (первый оператор, вхождение №4), (второй оператор, вхождение №3), …
C: (первый оператор, вхождение №5), (второй оператор, вхождение №1), (третий оператор, вхождение №5).

3.1. Проверяем, является ли переменная счетчиком данного цикла, если да, то переходим к следующей переменной. В рассматриваемом примере i-переменная цикла и для нее никаких дальнейших действий предприниматься не будет. Дальнейшая часть алгоритма будет выполняться только для переменной C.

3.2. Проверяем, есть ли вхождения данной переменной в правой части операторов присваивания тела данного цикла. Для этого проходим по вхождениям данной переменной и смотрим, есть ли вхождения с номером 1. В рассматриваемом примере переменная C входит в левую часть второго оператора присваивания (т.е. вхождение имеет номер 1).

3.3. Находим блок (список операторов), являющийся верхним для оператора цикла. С каждым блоком в дереве идентификаторов связанно подпространство имен. Создаем в этом подпространстве массив с

уникальным именем (в основе имени лежит имя скалярной переменной). Размер массива устанавливаем равным размеру цикла плюс 1.

В данном примере получаем:

```
{
    int C;
    int C_1[11];
    ………………
    for (i=0; i<10; i++)
    {
      ……………
    }
    ………………
}
```

3.4. Вставляем перед телом цикла оператор присваивания, в котором первый элемент массива инициализируется значением скалярной переменной. В рассматриваемом случае перед телом цикла будет вставлен следующий оператор присваивания:

```
C_1[0]=C;
```

3.5. В теле цикла заменяем все вхождения переменной на вхождения массива. При этом, если это вхождение предшествует первому появлению данной скалярной переменной в качестве генератора, то в нем используется значение переменной, полученное на прошлой итерации цикла (для первой итерации цикла будет использоваться значение, которое переменная имела до входа в цикл). Поэтому индекс элемента массива выставляется равным счетчику цикла. Для элемента массива, заменяющего первое вхождение скаляра в качестве генератора, индекс выставляется равным счетчику цикла плюс 1. Всем последующим вхождениям элементов массива также приписывается индекс, равный счетчику цикла плюс 1, т.к. для исходных скаляров использовалось значение переменной, полученное на этой же итерации.

Для данного примера получим следующий видоизмененный цикл:

```
for (i=0; i<10; i++)
{
  A[i]=B[i]+C_1[i];
  C_1[i+1]=A[i-1];
  B[i]=A[i]+C_1[i+1];
}
```

3.6. Вставляем после тела цикла оператор присваивания, в котором скалярной переменной возвращается значение последнего элемента массива. В рассматриваемом примере после тела цикла будет вставлен следующий оператор присваивания:

```
C=C_1[10];
```

В итоге преобразованный фрагмент программы будет иметь вид:

```
{
  int C;
  int C_1[11];
  ..................
  C_1[0]=C;
  for (i=0; i<10; i++)
  {
    A[i]=B[i]+C_1[i];
    C_1[i+1]=A[i-1];
    B[i]=A[i]+C_1[i+1];
  }
  C=C_1[10];
  ..................
}
```

Полезность данного вспомогательного преобразования также видна на данном примере:

```
for (i=0; i<10; i++)
{
  A[i]=B[i]+C;

  C=A[i-1];

  B[i]=A[i]+C;
}
```

По рисунку видно зависимости порожденные переменной C препятствуют разрезанию. Применив предварительно растягивание скаляров, получим цикл:

```
for (i=0; i<10; i++)
{
  A[i]=B[i]+C_1[i];

  C_1[i+1]=A[i-1];

  B[i]=A[i]+C_1[i+1];
}
```

В нем уже нет некоторых дуг информационной зависимости и его можно разрезать:

```
C_1[0]=C;
for (i=0; i<10; i++)
```

```
{
  A[i]=B[i]+C_1[i];
  C_1[i+1]=A[i-1];
}

for (i=0; i<10; i++)
  B[i]=A[i]+C_1[i+1];
C=C_1[10];
```

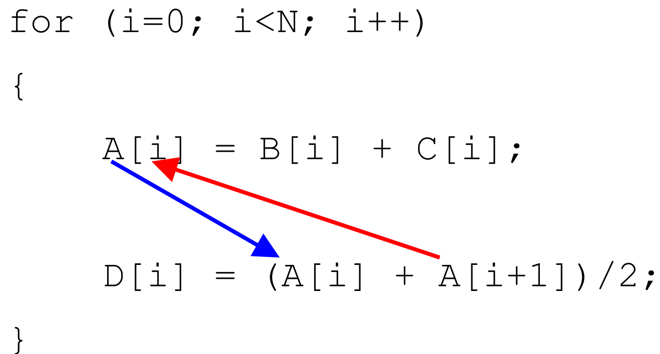
Растягивание скаляров является частным случаем увеличения размерности массива, преобразования, увеличивающего размерность массива на 1.

2.4.2 Расщепление вершин (Введение временных массивов)

Это преобразование позволяет удалять в программных циклах дуги антизависимости (может быть заменяя их другими). Суть преобразования в том, чтобы какое-нибудь вхождение приравнять к новой переменной и заменить этой переменной. Данное вспомогательное преобразование реализовано в ОРС Морылевым Романом.

В следующем цикле есть дуга антизависимости от вхождения A[i+1] к первому вхождению A[i]. Отметим, что эту дугу нельзя устранить перестановкой операторов тела цикла местами (такая перестановка нарушит равносильность программы ввиду наличия дуги потоковой зависимости между двумя вхождениями A[i]).

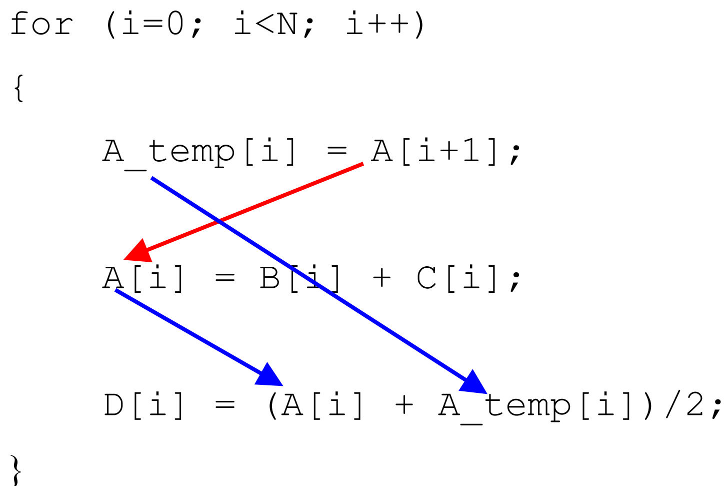
Пример 2



```
for (i=0; i<N; i++)
{
     A[i] = B[i] + C[i];

     D[i] = (A[i] + A[i+1])/2;
}
```

Данный цикл равносилен следующему:



```
for (i=0; i<N; i++)
{
     A_temp[i] = A[i+1];

     A[i] = B[i] + C[i];

     D[i] = (A[i] + A_temp[i])/2;
}
```

После выполнения данного преобразования можно цикл разбить на три части (чему мешала прежде дуга антизависимости, ведущая «снизу-вверх»):

```
for (i=0; i<N; i++)
     A_temp[i]=A[i+1];
for (i=0; i<N; i++)
     A[i]=B[i]+C[i];
for (i=0; i<N; i++)
     D[i]=(A[i]+A_temp[i])/2;
```

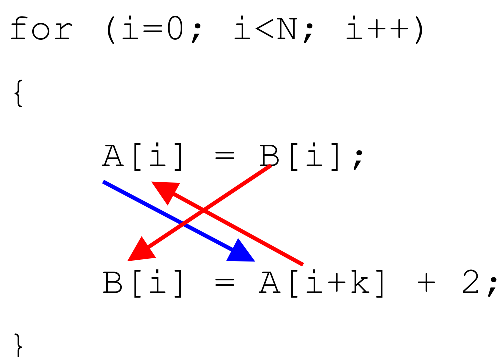
2.4.3 Применение динамического программирования

Не всегда на этапе компиляции по тексту фрагмента программы можно понять, насколько хорошо этот фрагмент можно распараллелить (оптимизировать). Препятствием могут служить неопределенные значения внешних переменных (например, в индексных выражениях). Иногда в таких случаях можно сформировать предикат, проверяющий возможность применения нужного преобразования. Тогда исходный фрагмент программы можно заменить новым фрагментом, проверяющим значение этого предиката и, в случае истинности предиката, выполняющим действия одним способом (параллельно), а в случае ложности – другим.

Приведем пример использования методов динамического программирования для преобразования разрезания.

Пример 3

Следующий цикл нельзя разрезать непосредственно, так как на этапе компиляции неизвестно значение переменной k, поэтому на графе информационных зависимостей будут представлены обе дуги:



Если $k \leq 0$, то между вхождениями массива A остается только дуга потоковой зависимости, если $k>0$ – то антизависимости. В первом случае цикл разрезать можно, а во втором – нельзя.

Заменим данный фрагмент программы следующим:

```
if (k<=0)
    for (i=0; i<N; i++)
    {
     A[i] = B[i];

     B[i] = A[i+k] + 2;
    }
else
    for (i=0; i<N; i++)
    {
          A[i] = B[i];

          B[i] = A[i+k] + 2;
     }
```

теперь первый цикл можно разрезать:

```
if (k<=0)
    for (i=0; i<N; i++)
          A[i] = B[i];
    for (i=0; i<N; i++)
          B[i] = A[i+k] + 2;
else
     ……………
```

2.5. Вынос инварианта из цикла

Суть данного преобразования (называемого также «чистка вверх» [16]) заключается в вырезании инварианта в отдельный цикл и удалении заголовка

цикла для него. Инвариантом называется не рекуррентный оператор присваивания, левая часть которого не зависит от счетчика цикла. При этом в нем не должно быть вызовов функций.

Пример 1

В следующем примере:

```
for (i=0; i<N; i++)
    B=A[i]+i+1;
```

оператор в теле цикла является инвариантом. Значение переменной B по окончании цикла можно выразить, подставив в правую часть вместо счетчика цикла значение левой границы цикла. Для данного примера получаем:

```
B=A[N-1]+N-1+1;
```

Этот оператор эквивалентен исходному циклу.

В рамках данной работы была получена программная реализация данного преобразования. InvariantExportation входит в пакет оптимизирующих и распараллеливающих преобразований ОРС.

Алгоритм реализованного преобразования состоит из следующих шагов:

1) Проверяем левую часть на независимость от счетчика цикла.

2) Проверяем оператор присваивания на отсутствие рекуррентности (т.е. отсутствие дуг в графе Лампорта ведущих из правой части в левую).

3) Проверяем оператор на отсутствие вызовов функций

4) Копируем данный оператор из тела цикла после него.

5) Заменяем в правой части все вхождения счетчика цикла на правую границу цикла.

6) Удаляем исходный цикл.

Пример 2. Фрагмент

```
for (i=0; i<N; i++)
    A[k]=5+B[i];
```

эквивалентен

```
    A[k]=5+B[N-1];
```

Пример 3. К циклу:

```
for (i=0; i<N; i++)
    A[i]=B[i]+1;
```

преобразование применить нельзя, поскольку левая часть оператора присваивания зависит от счетчика цикла.

Пример 4. К циклу:

```
for (i=0; i<N; i++)
    A=A+k;
```

преобразование применить нельзя, оператор в теле цикла рекурсивный.

В случае, когда тело цикла содержит более одного оператора, преобразование сводится к следующим действиям:

1) разрезаем данный цикл на максимальное число частей

2) для полученных циклов, тела которых содержат один оператор присваивания, пытаемся применить вынесение инварианта. Если после разрезания один из полученных циклов имеет более одного оператора в теле, это означает, что между его операторами существуют перекрестные

зависимости, поэтому ни один из операторов данного цикла не может быть инвариантом.

3) пытаемся, используя преобразование перестановки, вынести все полученные инварианты до или после циклов, являющихся результатом разрезания.

4) попарно сливаем оставшиеся циклы.

Пример 5. В данном цикле:

```
for (i=0; i<N; i++)
{
   A[i]=B[i]+1;
   C[k]=2*A[i];
   B[i]=D[i]-i;
}
```

второй оператор является инвариантом. Разрезав данный цикл на три и применив к каждому из полученных преобразование вынесения, получим:

```
for (i=0; i<N; i++)
   A[i]=B[i]+1;

C[k]=2*A[N-1];

for (i=0; i<N; i++)
   B[i]=D[i]-i;
```

Теперь инвариант можно переставить с третьим циклом и слить первый и второй циклы:

```
for (i=0; i<N; i++)
{
    A[i]=B[i]+1;
    B[i]=D[i]-i;
}
C[k]=2*A[N-1];
```

3 Слияние программных циклов

Данное преобразование заменяет фрагмент программы, состоящий из двух подряд написанных циклов с одинаковыми заголовками, одним циклом (loop fusion [12], [13], [14]).

Это преобразование является обратным к разбиению циклов.

Была получена программная реализация данного преобразования LoopsMerging.

Пример 1

Данные два цикла:

```
for (i=0; i<=9; i=i+1)
  A[i]=B[i]+C;
for (i=0; i<=9; i=i+1)
  E[i]=A[i]*2+D[i];
```

можно эквивалентно слить в один:

```
for (i=0; i<=9; i=i+1)
{
  A[i]=B[i]+C;
  E[i]=A[i]*2+D[i];
}
```

Пример 2

Данные два цикла:

```
for (i=0; i<=9; i=i+1)
  A[i]=B[i]+C;
for (i=0; i<=9; i=i+1)
  D[i]=A[i+1]+2;
```

эквивалентно слить в один нельзя, так как в исходном примере во втором цикле используются значения A[i+1], уже посчитанные в первом цикле, а в слитом цикле:

```
for (i=0; i<=9; i=i+1)
{
  A[i]=B[i]+C;
  D[i]=A[i+1]+2;
}
```

во втором операторе используются значения A[i+1] еще не посчитанные.

3.1 Область использования слияния

Слияние применяется для однопроцессорных и конвейерных машин.

Для однопроцессорных машин данное преобразование применяется как оптимизирующее, так как оно уменьшает количество проверок условия выхода из цикла.

Также слияние применяется для конвейерных машин . В случае, когда тело цикла содержит меньше простых операций (+, -, * и т.п.), чем

элементарных элементов конвейера, слияние может помочь достичь в объединенном цикле необходимое количество операций. Возможны также комбинации преобразований разрезания и слияния для достижения необходимого результата.

3.2 Условия корректности и эквивалентности преобразования

Если после выполнения преобразования получается цикл, который можно, сохраняя равносильность, разрезать на две части, получив исходный фрагмент, то данное преобразование равносильно.

Однако, есть ограничение, связанное с синтаксической корректностью. В исходной программе не должно быть переходов на заголовок второго цикла, поскольку в этом случае в результирующей программе этот переход исчезнет или возникнет недопустимый синтаксисом переход извне во внутрь цикла. Кроме того, оба цикла должны находиться в одном блоке операторов.

3.3 Алгоритм преобразования

1) Проверяем, являются ли входные циклы канонизированными;
2) проверяем, находятся ли оба цикла в одном блоке и следуют ли друг за другом;
3) сравниваем правые границы циклов (они должны быть одинаковые);
4) создаем временный цикл, в который копируем операторы первого и второго циклов;
5) строим граф информационных зависимостей Лампорта по телу данного цикла;
6) проверяем, есть ли зависимости, ведущие из второй части операторов в первую если таких зависимостей нет, то преобразование корректно. В этом случае удаляем исходные циклы, иначе удаляем временный цикл.

Пример 1

Опишем алгоритм более подробно на следующем простом примере:

```
{
  ..............
  for (i=0; i<=9; i=i+1)
    A[i]=B[i]+1;
  for (i=0; i<=9; i=i+1)
    C[i]=A[i];
  ..............
}
```

1) Проверяем оба цикла на каноничность, т.е. проверяем, что левая граница равна 0, правая граница - константа и шаг равен 1. Для рассматриваемого примера это выполняется.

2) Проверяем, что верхние (содержащие их) блоки для обоих циклов совпадают. Также проверяем, что оба цикла идут подряд. В данном примере это выполняется.

3) Сравниваем (как выражения) правые границы циклов. В данном примере у обоих циклов правая граница равна 9.

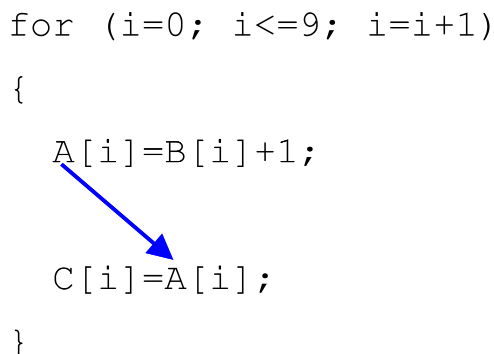
4) Создаем временный цикл, содержащий операторы первого и второго исходных циклов. Для этого копируем первый цикл, а затем в конец тела этого временного цикла копируем операторы второго цикла.

Для рассматриваемого примера получаем следующий временный цикл:

```
for (i=0; i<=9; i=i+1)
{
  //Первая часть операторов
  A[i]=B[i]+1;
  //Вторая часть операторов
  C[i]=A[i];
}
```

5) Строим граф информационных зависимостей Лампорта для временного цикла.

В рассматриваемом случае граф содержит одну дугу потоковой зависимости между вхождениями массива A в первом и втором операторах присваивания:



6) Проверяем, есть ли дуги, ведущие из второй части операторов в первую. Если такие дуги есть, то преобразование не эквивалентно. В данном примере таких дуг нет, поэтому преобразование корректно. Удаляем исходные два цикла. Теперь временный цикл и будет искомым слитым циклом:

```
{
  ...............
  for (i=0; i<=9; i=i+1)
  {
    A[i]=B[i]+1;
    C[i]=A[i];
  }
  ...............
}
```

3.4 Случаи сложных преобразований

Иногда преобразование слияния можно применить в контексте с несколькими другими преобразованиями, такими как разрезание циклов и отщепление итераций.

Пример 1. Следующие циклы:

```
for (i=0; i<90;  i++)
  A[i]=i;
for (i=0; i<100; i++)
  B[i]=A[i]+2;
```

слить нельзя, поскольку они имеют различные правые границы. Однако пространство итераций второго цикла в данном случае можно эквивалентно разбить на две части:

```
for (i=0;  i<90;  i++)
  A[i]=i;
```

```
for (i=0;  i<90;  i++)
  B[i]=A[i]+2;
for (i=90; i<100; i++)
  B[i]=A[i]+2;
```

Теперь первые два цикла можно слить:

```
for (i=0;  i<90;  i++)
{
  A[i]=i;
  B[i]=A[i]+2;
}
for (i=90; i<100; i++)
  B[i]=A[i]+2;
```

Пример 2

Следующие два цикла:

```
for (i=0; i<100; i++)
  A[i]=B[i]+1;
for (i=0; i<100; i++)
  C[i]=A[i+1]*2;
```

слить непосредственно нельзя. Этому мешает дуга антизависимости, ведущая из второго вхождения A[i+1] в первое вхождение A[i].

Применим к данным циклам преобразование отщепления итераций. Отщепим от первого цикла первую итерацию, а от второго цикла последнюю итерацию:

```
A[0]=B[0]+1;
for (i=1; i<100; i++)
  A[i]=B[i]+1;

for (i=0; i<99; i++)
  C[i]=A[i+1]*2;
C[99]=A[100];
```

Канонизируем первый цикл, для этого сдвинем его пространство итераций так, чтобы левая граница цикла была равна 0:

```
A[0]=B[0]+1;
for (i=0; i<99; i++)
  A[i+1]=B[i+1]+1;
for (i=0; i<99; i++)
  C[i]=A[i+1]*2;
C[99]=A[100];
```

Теперь циклы можно корректно слить:

```
A[0]=B[0]+1;
for (i=0; i<99; i++)
{
  A[i+1]=B[i+1]+1;
  C[i]=A[i+1]*2;
}
C[99]=A[100];
```

Пример 3

Для следующих двух циклов:

```
for (i=0; i<100; i++)
  A[i]=B[i]+1;
for (i=0; i<100; i++)
{
  C[i]=A[i+1]*2;
  D[i]=B[i]-2;
}
```

применить слияние нельзя из-за антизависимости между вхождениями массива A. В данном случае можно применить перестановку операторов из первого во второй. Для этого второй цикл разрезается на два:

```
for (i=0; i<100; i++)
  A[i]=B[i]+1;
for (i=0; i<100; i++)
  C[i]=A[i+1]*2;
for (i=0; i<100; i++)
  D[i]=B[i]-2;
```

Затем применяется преобразование перестановки (описанное дальше) ко второму и третьему циклу:

```
for (i=0; i<100; i++)
  A[i]=B[i]+1;
for (i=0; i<100; i++)
  D[i]=B[i]-2;
for (i=0; i<100; i++)
  C[i]=A[i+1]*2;
```

Теперь первые два цикла можно слить:

```
for (i=0; i<100; i++)
{
  A[i]=B[i]+1;
  D[i]=B[i]-2;
}
for (i=0; i<100; i++)
  C[i]=A[i+1]*2;
```

4 Выделение векторизуемых циклов

Некоторые циклы на векторных машинах могут выполняться с помощью одной (векторной) команды, т.е. тело цикла выполняется одновременно для всех итераций. Для выявления таких циклов в рамках данной работы в ОРС была реализована функция нахождения векторизуемых циклов.

Предварительное применение преобразования разрезания цикла может увеличить число векторизуемых циклов.

4.1 Примеры векторизуемых циклов

Программный цикл является векторизуемым, если в его теле нет циклически зависимых зависимостей, т.к. в противном случае следующие итерации нельзя выполнять до того, как не будут выполнены предыдущие. Кроме того, операторы тела цикла не должны содержать вызовов функций.

Пример 1

```
for (i=0; i<1000; i++)
  A[i]=B[i]+2*i;
```

Данный цикл векторизуемый и может быть заменен одной векторной командой.

Пример 2

```
for (i=0; i<1000; i++)
  A[i]=A[i-1]*2;
```

Данный цикл не векторизуемый, так как оператор присваивания в его теле рекуррентен (нельзя посчитать, например, вторую итерацию цикла до того, как была выполнена первая).

4.2 Функция нахождения векторизуемых циклов

В ОРС была реализована функция автоматического определения векторизуемых циклов.

Алгоритм работы функции:

1) Проходим по телу цикла и проверяем его на отсутствие вызовов функций.

2) Строим граф информационных зависимостей Лампорта по телу цикла.

3) Проверяем по нему цикл на отсутствие циклически порожденных зависимостей.

4) Если все предыдущие пункты выполнены, то помечаем данный цикл как векторизуемый.

Кроме этого, была написана функция автоматического нахождения всех векторизуемых циклов в блоке. Она рекурсивно проходит программный блок

и для всех найденных циклов вызывает функцию определения векторизуемых циклов.

5 Исследование разрезаемости случайных циклов

Для тестирования применимости преобразования разрезания к различным случайным циклам, автором работы была написана специальная программа. Она входит в проект ОРС под названием TestSplitting. Программа состоит из двух частей: генератора случайных циклов и сборщика статистики.

Целью данной программы является сбор статистической информации о разрезаемости циклов, имеющих различные параметры, и исследование эффективности вспомогательных преобразований. Неразрезаемость цикла означает, что в нем существуют рекурсивные зависимости. В качестве параметров цикла рассматривались следующие величины: количество операторов цикла, количество массивов и скаляров, входящих в операторы цикла, длина правых частей и т.п.

Данная информация представляет большой интерес для разработчиков распараллеливающих компиляторов. Она помогает определить полезность применения в компиляторе тех или иных вспомогательных преобразований. Полученная информация также может оказаться полезной при разработке новых суперкомпьютеров, т.к. можно настроить архитектуру суперЭВМ на эффективное выполнение наиболее часто встречающихся неразрезаемых циклов.

Модуль генерации случайных циклов может использоваться также для тестирования других преобразований, работающих с циклами. При этом тестируются корректность преобразования (не завершается ли программа ошибкой), время выполнения преобразования и его эффективность.

5.1 Генерация случайных циклов

Для тестирования преобразований, работающих с циклами, в рамках данной работы была написана программа генерации случайных циклов во внутреннем представлении ОРС.

В качестве входных параметров процедура генерации получает следующие данные о создаваемом цикле:

1) минимальное и максимальное число используемых теле цикла массивов;

2) минимальное и максимальное число используемых теле цикла скалярных переменных;

3) минимальное и максимальное число операторов присваивания в теле цикла;

4) максимальное для правой части операторов присваивания (число вхождений массивов и скалярных переменных). Минимальная длина правой части равна 1;

5) максимальное значение константы в индексе вхождения массива (константа c – во вхождении A[i±c]).

Программа генерирует во внутреннем представлении блок. В нем создаются массивы и скалярные переменные, число которых случайно и находится в заданных пределах. В блоке создается цикл for. Далее в тело цикла добавляется случайное число операторов присваивания (их число также находится в заданных пределах). У каждого оператора присваивания случайным образом выбирается длина правой части (не более максимальной). Затем на место каждого вхождения случайным образом подставляет один из описанных циклов или скалярных переменных. При этом константа, прибавляемая или отнимаемая от индекса массива, также выбирается случайно.

В индексных выражениях чаще всего эта константа равна 0 (A[i]), константа, равная 1, встречается реже, 2 – еще реже и т.д. Поэтому было принято решение использовать для выбора константы не обычный генератор случайных чисел, а модернизированный. Он возвращает случайную величину σ^n, где σ - случайная величина, равномерно распределенная на интервале (0, 1), $n>1$. Понятно, что при таком распределении числа, близкие к 0, будут встречаться чаще, чем числа, близкие к 1. Поэтому в сгенерированных индексах выражение [i] будет встречаться чаще, чем выражение [i+m], где m – максимальное значение константы.

Пример 1

Приведем пример сгенерированного случайного цикла со следующими параметрами: число используемых массивов равно 3, скалярных переменных - 1, операторов в теле цикла - 3, максимальная длина правой части – 3, максимальное значение константы в индексном выражении – 2.

```
int i;
int A[100];
int B[100];
int C[100];
int D;
for (i=0; i<=99; i+=1)
{
  A[i]=(B[i-1]+D);
  C[i+1]=(D+B[i]);
  B[i]=(D+A[i]);
}
```

5.2 Результаты исследования разрезаемости случайных циклов

Для тестирования разрезаемости случайных циклов использовалась генерация случайных циклов с различными параметрами. Для этого многократно генерировались случайные циклы, а затем они подавались на вход преобразования разрезания. Далее подсчитывалось общее число сгенерированных циклов (с данным числом операторов), и число циклов, которые разрезать не удалось. В результате был получен процент неразрезаемых циклов для каждого набора параметров.

Затем к неразрезаемым случайным циклам по очереди применялось вспомогательные преобразования (растягивание скаляров и введение временных массивов) и отслеживалось, насколько данные преобразования эффективны, т.е. какой процент циклов стал разрезаться после данного вспомогательного преобразования.

Число протестированных случайных циклов с каждым набором параметров равнялось 10000. В результате проведенного анализа были получены следующие статистические данные:

1) статистика зависимости неразрезаемости случайных циклов от числа используемых массивов:

Таблица 1 (число операторов - 2, максимальна длина правой части - 3, максимальное значение константы в индексных выражениях – 2, без скалярных переменных)

Число массивов	**Процент неразрезаемых циклов**
1	66%
2	37%
3	23%
4	15%
5	11%

Таблица 2 (число операторов - 3, максимальна длина правой части - 3, максимальное значение константы в индексных выражениях – 2, без скалярных переменных)

Число массивов	Процент неразрезаемых циклов
1	69%
2	39%
3	21%
4	12%
5	7%

Из таблиц 1 и 2 видно, что при увеличении числа массивов, процент неразрезаемых циклов уменьшается.

2) статистика зависимости неразрезаемости циклов от числа операторов присваивания в теле цикла:

Таблица 3 (число используемых массивов - 3, максимальна длина правой части - 3, максимальное значение константы в индексных выражениях – 2, без скалярных переменных)

Число операторов в теле цикла	Процент неразрезаемых циклов
2	23%
3	21%
4	26%
5	31%
6	37%
7	44%

Из таблицы 3 видно, что при увеличении числа операторов в теле цикла, процент неразрезаемых циклов растет. Это обуславливается тем, что при

увеличении числа операторов возникает больше рекуррентных связей, стягивающих операторы в одну компоненту сильной связности. При этом, с увеличением максимальной длины правой части интенсивность роста процента неразрезаемых циклов растет. Это видно из таблиц 4 и 5:

Таблица 4 (число используемых массивов - 3, максимальна длина правой части - 2, максимальное значение константы в индексных выражениях – 2, без скалярных переменных)

Число операторов в теле цикла	**Процент неразрезаемых циклов**
2	19%
3	13%
4	20%
5	24%
6	29%
7	35%

Таблица 5 (число используемых массивов - 4, максимальна длина правой части - 4, максимальное значение константы в индексных выражениях – 2, без скалярных переменных)

Число операторов в теле цикла	**Процент неразрезаемых циклов**
2	28%
3	30%
4	35%
5	42%
6	37%
7	44%

Приведем график зависимости неразрезаемости циклов от числа операторов тела цикла и числа используемых массивов:

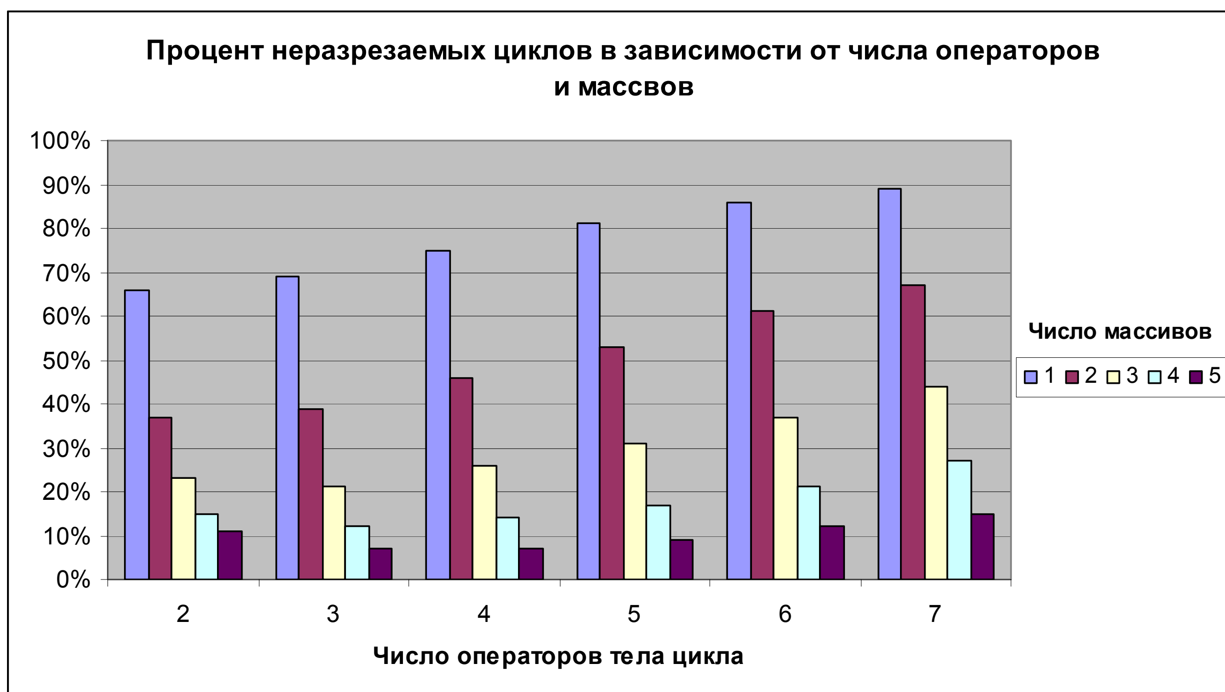


Рис. 1

По графику видно, что неразрезаемость цикла зависит в большей степени от числа используемых массивов.

При тестировании также исследовалось время выполнения отдельных блоков распараллеливающего компилятора. Результат представлен в таблице 2, в правой колонке которой указан процент выполнения данного блока относительно общего времени выполнения программы.

Таблица 6

Генерация случайного цикла	7,9%
Построение графа информационных зависимостей Лампорта	84,3%
Разрезание цикла	4,7%

Из таблицы 6 видно, что наибольшее время работы программы занимает построение графа Лампорта.

Заключение

В данной работе были получены следующие результаты:

– Программно реализованы алгоритмы разрезания и слияния программных циклов.

– Программно реализовано преобразование растягивание скаляров.

– Написан ряд редакторских преобразований во внутреннем представлении ОРС.

– Написан депарсер из внутреннего представления ОРС в язык С.

– Программно реализовано преобразование «вынесение инварианта из цикла».

– Написана функция нахождения векторизуемых циклов.

– Написана программа генерации случайных циклов во внутреннем представлении ОРС. Проведено исследование частоты разрезаемости случайных программных циклов и эффективности вспомогательных преобразований.

Направления дальнейших исследований:

– Программная реализация дополнительных вспомогательных преобразований, способствующих разрезанию, таких как переименование переменных, отщепление итераций цикла и т.п.

– Разработка динамически версий распараллеливающих преобразований разрезание и слияние циклов.

– Разработка преобразований способствующих слиянию программных циклов.

– Разработка системы автоматического распознавания необходимости применения вспомогательных преобразований на основе использования более точного фактор-графа по компонентам сильной связности графа вхождений переменных (а не операторов).

– Сбор статистики разрезаемости циклов на реальных программах.

Литература

1. Векторизация программ. // Векторизация программ: теория, методы, реализация. / Сборник переводов статей М.: Мир, 1991. С. 246 - 267.

2. Штейнберг Б.Я.. Математические методы распараллеливания рекуррентных циклов для суперкомпьютеров с параллельной памятью. Ростов-на-Дону, 2004.

3. Штейнберг Б.Я., Арутюнян О.Э., Бутов А.Э., Гуфан К.Ю., Морылев Р., Науменко С.А., Петренко В.В., Тузаев А., Черданцев Д.Н., Шилов М.В., Штейнберг Р.Б., Шульженко А.М. Обучающая распараллеливанию программа на основе ОРС.// Научно-методическая конференция «Современные информационные технологии в образовании: Южный федеральный округ», Ростов-на-Дону, 12-15 мая 2004 г., с. 248-250.

4. Штейнберг Б.Я., Черданцев Д.Н., Науменко С.А., Бутов А.Э., Петренко В.В. Преобразования программ для открытой распараллеливающей системы// Искусственный интеллект. Научно-теоретический журнал. Институт проблем искусственного интеллекта НАНУ. Украина, Донецк, ДонДИШИ, “Наука и Освита”, 2003, № 3, с. 97-104.

5. Евстигнеев В.А., Касьянов В.Н. Оптимизирующие преобразования в распараллеливающих компиляторах// Программирование, 1996, № 6, с. 12-26.

6. Касьянов В.Н., Оптимизирующие преобразования программ/ М., «Наука», 1988 г., 336 с.

7. Воеводин В.В. Воеводин Вл.В. Параллельные вычисления, С-Петербург «БХВ-Петербург», 2002, 599 с.

8. Воеводин В.В. Пакулев В.В. Определение дуг графа алгоритма. - М., Отдел вычислительной математики АН СССР, 1989, 22 с. (препринт).

9. Lamport L. The parallel execution of DO loops// Commun. ACM.- 1974.- v.17, N 2, p. 83-93.

10. Штейнберг Б.Я., Бутов А.Э., Науменко С.А., Петренко В.В., Черданцев Д.Н., Штейнберг Р.Б., Шульженко А.М. Полуавтоматическое

распараллеливание на основе Открытой распараллеливающей системы. // Сборник трудов всероссийской научно-технической конференции «Параллельные вычисления в задачах математической физики». Ростов-на-Дону. Ростовский государственный университет, 2004, с. 207.

11. Штейнберг Б.Я. Разбиение циклов для исполнения на суперкомпьютере со структурой перестраиваемого конвейера. // Искусственный интеллект. Научно-теоретический журнал. Институт проблем искусственного интеллекта НАНУ. Украина, Донецк, ДонДИШИ, “Наука и Освита”, 2002, № 3 , с. 331-338.

12. Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, Amy W. Lim. An Overview of a Compiler for Scalable Parallel Machines. // Computer Systems Laboratory Stanford University, CA 94305.

13. Naraig Manjikian, Tarek S. Abdelrahman. Fusion of Loops for Parallelism and Locality. // Department of Electrical and Computer Engineering, The University of Toronto, Toronto, Ontario, Canada.

14. Dattatraya Kulkarni, Stumm M. Loop and Data Transformations: A Tutorial // Department of Computer Science and Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada.

15. Штейнберг Б.Я., Макошенко Д.В., Черданцев Д.Н., Шульженко А.М. Внутреннее представление в открытой распараллеливающей системе// Искусственный интеллект. Научно-теоретический журнал. Институт проблем искусственного интеллекта НАНУ. Украина, Донецк, ДонДИШИ, “Наука и Освита”, 2003, № 3 , с. 89-96.

16. Касьянов В.П. Оптимизирующие преобразования программ. // М.: Гл. ред. Физ.-мат. Лит., 1988. – 336 с.