

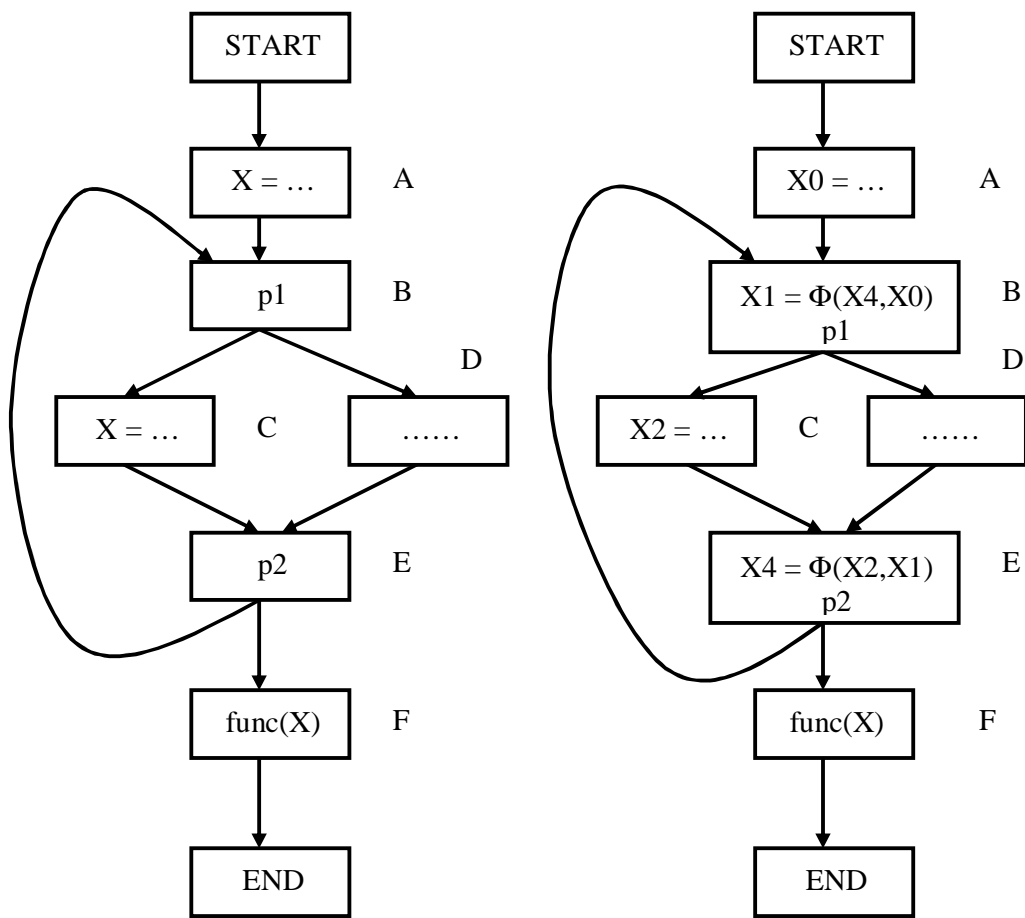
Эффективный алгоритм построения формы статического единственного присваивания

Дроздов А. Ю., Новиков С. В.

Институт микропроцессорных вычислительных систем РАН
sasha@mcst.ru, novikov@mcst.ru

Введение

Форма статического единственного присваивания (Static Single Assignment, SSA) является одной из самых распространенных форм представления потока данных программы и активно используется в большинстве современных оптимизирующих компиляторов [1].



а) исходный управляющий граф

б) управляющий граф и ϕ -функции

Рис. 1 Программа и ее SSA форма.

Ключевым шагом в ее построении является нахождение мест размещения ϕ -функции в предположении, что задано множество узлов управляющего графа, которые содержат записи в некоторую переменную программы. Напомним, что в программе, представленной в SSA-форме, любая переменная может быть определена только один раз. Для того чтобы соблюсти это ограничение и, тем самым, перевести программу в SSA-форму, необходимо выполнить следующие действия:

- Разместить в точках схождения потока управления ϕ -узлы. ϕ -узел для некоторой переменной – это операция, выбирающая среди множества значений переменной нужное.
- Переименовать все переменные, так чтобы каждому определению соответствовала своя, уникальная переменная.

На рис. 1 приведен пример преобразования программы в SSA форму. В точках схождения (узлы B и E) построены ϕ -узлы, а затем проведено переименование переменной X.

Проблема нахождения мест размещения ϕ -функции сводится к вычислению итерационного фронта доминирования (Iterated Dominance Frontier, IDF) [2] для множества узлов управляющего графа, содержащих операции записи в переменную, для которой строится SSA-форма. Напомним, что IDF некоторого линейного участка N определяется как объединение IDF линейных участков, входящих в DF N. Линейный участок N' входит в DF N, если N доминирует над одним из предшественников N', но не доминирует (строго) над N'. При построении SSA-формы для программы эта процедура применяется ко всем переменным, участвующим в построении. В настоящий момент разработано много алгоритмов нахождения IDF для заданного множества узлов управляющего графа [2-4], на основе которых эффективно решается задача построения SSA-формы для отдельной переменной.

В работе сделан обзор известных алгоритмов нахождения множества IDF, имеющих наилучшую временную асимптотическую оценку и предложена их модификация, позволяющая существенно ускорить работу алгоритмов построения ϕ -функций для множества переменных и как следствие ускорить построение SSA-формы для всей программы. В [4] проведена классификация опубликованных алгоритмов, обладающих наилучшими показателями временной сложности. Каждый алгоритм обеспечивает нахождение мест для ϕ -функций для одной переменной за линейное время $O(E)$, где E – число дуг управляющего графа. Для множества переменных сложность алгоритма имеет порядок $O(E)*V$, где V – количество переменных, для которых строится SSA-форма. Таким образом, количество переменных существенно влияет на скорость преобразования программы в SSA-форму. В работе предлагается метод, позволяющий уменьшать алгоритмическую сложность алгоритмов построения ϕ -функций. В новой оценке величина V заменяется на величину $V/\text{size}(\text{word})$, где $\text{size}(\text{word})$ – размер минимального элемента реализации пакета битовых векторов (в нашей реализации $\text{size}(\text{word})=32$). Метод использовался для ускорения работы алгоритма, спроектированного на основе работы [2], и реализованного в оптимизирующем компиляторе проекта “Эльбрус”. Для получения экспериментальных данных использовался пакет SPECint92 (www.spec.org).

1. Алгоритмы построения ϕ -функций

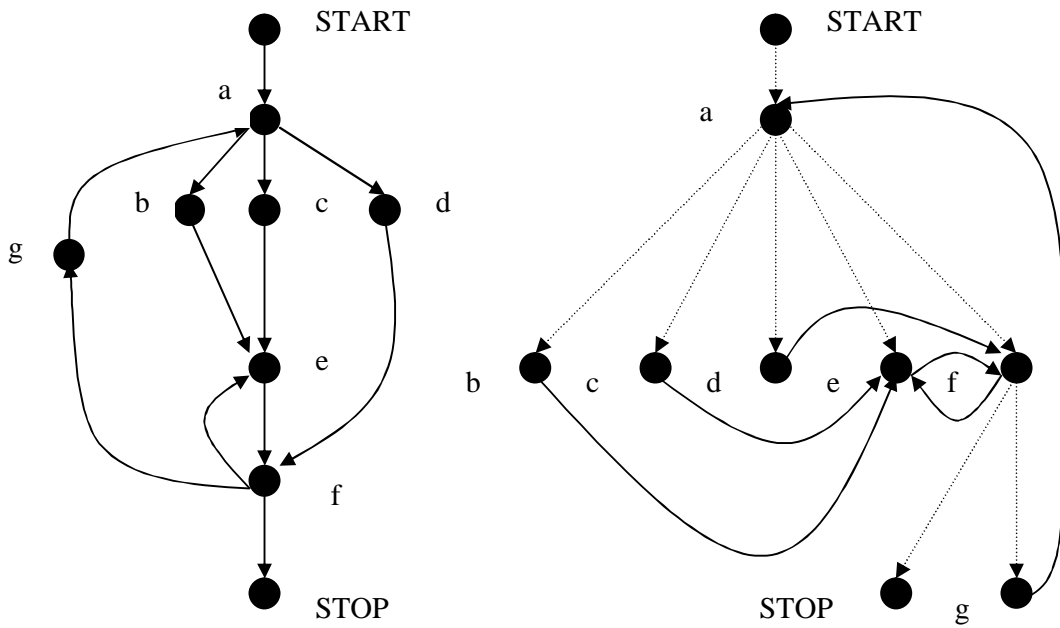
В алгоритмах, описанных в работах [2-4], проблема построения ϕ -функций решается на основании отношения DF. Если на DF отношении построить граф (DF-граф), то точками построения ϕ -функций для некоторого узла x являются все

достижимые из него в DF-графе узлы. Узел x входит во множество достижимых из себя узлов только при наличии ненулевого пути из x в x . Таким образом, задача построения ϕ -функций для исходного множества узлов S сводится к нахождению множества $IDF(S)$.

Сведение задачи построения ϕ -функций к задаче достижимости на DF-графе позволяет построить двухфазную схему нахождения $IDF(S)$. На первой фазе необходимо построить DF-граф на всем множестве узлов CFG. Время построения DF-графа оценивается как $O(|N|+|E|+|DF|)$, где $|DF|$ определяет время построения дуг DF-графа. Размер множества дуг DF-графа оценивается пропорционально квадрату количества узлов CFG ($|DF|=O(|N||N|)$). На второй фазе решается задача достижимости на DF-графе для исходного множества узлов S с нахождением множества $IDF(S)$. Задача достижимости решается за время $O(|V| + |DF|)$.

Другой класс алгоритмов построения ϕ -функций позволяет решить задачу за время $O(E)$, не строя явно DF-графа. Вычисления, аналогичные вычислению достижимости в DF графе, выполняются в этих алгоритмах итеративно, по мере вычисления DF отношений для текущих узлов. Алгоритмы работают локально, то есть вычисление DF отношений происходит не для всех узлов управляющего графа, как в двухфазном алгоритме, а только для необходимых узлов. При реализации алгоритмов используются дерево доминаторов и управляющий граф. На их основе строятся структуры данных, в терминах которых работают алгоритмы. Так, в [2] используется структура DJ-графа, а в алгоритмах [4] используются ω -DF-граф и структура данных, построенная на основе дерева доминаторов с добавлением некоторого класса дуг графа управления (*augmented dominator tree*, ADT). DJ-граф является частным случаем ADT.

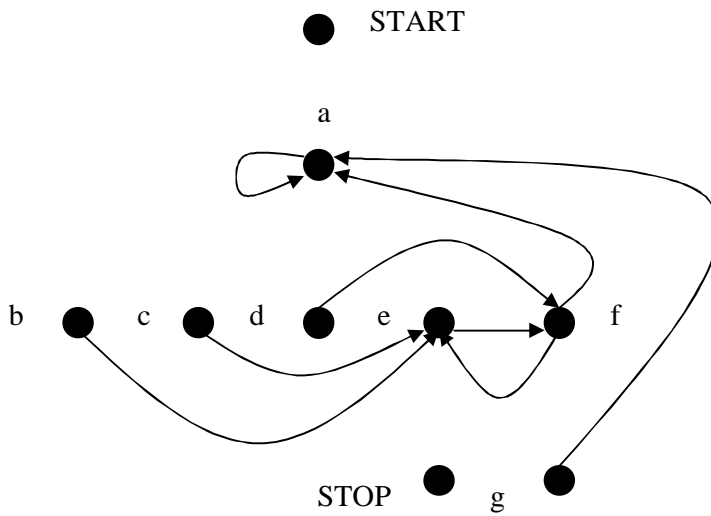
На рис. 2 изображен управляющий граф некоторой программы и соответствующие ему DF- и DJ-графы.



(a) Управляющий граф

(b) DJ-граф

.....> дуги дерева доминаторов
 —————> J-дуги



(c) DF-граф

Рис. 2. Управляющий граф, DJ-граф и DF-граф.

2. Решение задачи построения ϕ -функций для множества переменных

Все перечисленные алгоритмы решают задачу для случая одной переменной. Множество узлов управляющего графа, в которых есть присваивание в переменную, представляет собой исходное множество S для алгоритмов построения ϕ -функций этой переменной. Для другой переменной задача должна быть поставлена заново. Таким образом, сложность алгоритмов пропорциональна количеству переменных, для которых нужно строить ϕ -функции. Опишем метод ускорения алгоритмов построения ϕ -функций для множества переменных на примере двухфазного алгоритма. Идея ускорения состоит в переводе вычислений достижимости по DF-графу на битовые вектора (рис. 3).

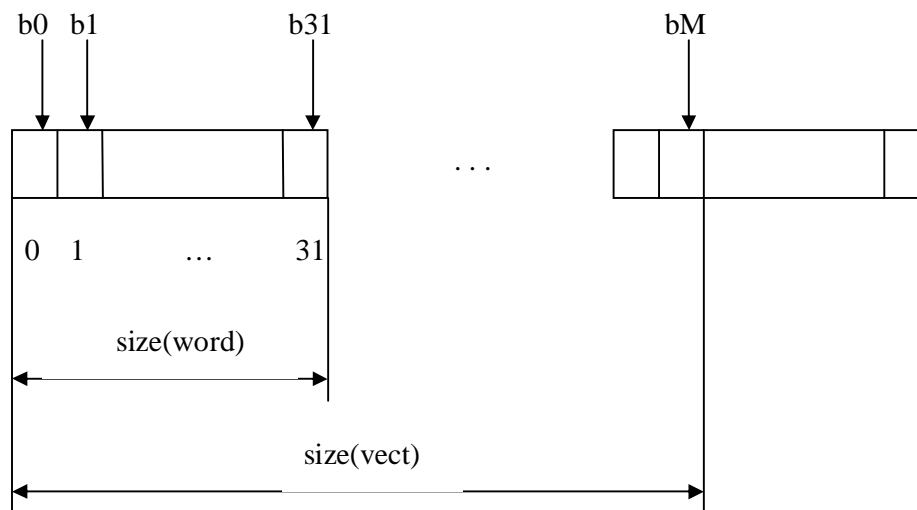


Рис. 3. Битовый вектор.

Битовым вектором является пронумерованный набор битов, над которым возможны следующие действия:

1. установка i -го бита в 0 или 1;
2. проверка i -го бита на 0 или 1;
3. конъюнкция битового вектора с другим битовым вектором того же размера (&);
4. дизъюнкция битового вектора с другим битовым вектором того же размера (!).

В реализации битовый вектор состоит из слов равной длины; размер слова влияет на скорость выполнения операций конъюнкции и дизъюнкции. Время выполнения этих операций прямо пропорционально количеству слов в битовом векторе, которое, в свою очередь, прямо пропорционально размеру вектора и обратно пропорционально размеру слова.

$$T(\&,!) = \text{size(vect)}/\text{size(word)} \quad (1)$$

Пусть задано начальное множество переменных V размера M . Перенумеруем все переменные исходного множества так, чтобы каждой соответствовал бит в битовых векторах, используемых в вычислениях достижимости (b_0, b_1, \dots, b_M). Поставим в соответствие каждому узлу DF-графа два битовых вектора: вектор присваиваний и вектор ϕ -функций. Подготовка исходных данных для работы алгоритма состоит в инициализации векторов присваиваний. Если в узле N есть присваивание в переменную b_i , то устанавливаем i -ый бит вектора присваиваний в 1. Задача распространения векторов на ациклическом графе решается линейным обходом графа. Для решения задачи линейным обходом на графе с циклами необходимо предварительно выделить на нем сильно связанные компоненты (Strongly Connected Component, SCC) [7] и редуцировать граф в ациклический вид, заменив SCC одним узлом, наследующим внешние дуги сильно связанных компонент. Алгоритм построения сильно связанных компонент имеет сложность $O(E)$, где E – количество дуг графа. Следует отметить, что для сводимого управляющего графа в DF-графе возможны лишь тривиальные циклы, то есть циклы из одного узла. Поэтому поиск сильно связанных компонент нужно осуществлять только в случае, если исходный управляющий граф был несводимым. Обход узлов графа при работе алгоритма выполняется в порядке RPO-нумерации узлов управляющего графа (reverse post order, [1]).

Опишем алгоритм распространения векторов на ациклическом графе.

Алгоритм:

$G=(N,E)$ – ациклический граф;

$Assign(N)$ – вектор присваиваний, соответствующий узлу N ;

$Placement(N)$ – вектор ϕ -функций, соответствующий узлу N ;

$Disjunct(vect1, vect2)$ – операция дизъюнкция битовых векторов.

Ниже дано описание алгоритма распространения векторов на ациклическом графе в терминах алгоритмического псевдоязыка, легко переводимого в любой современный язык, поддерживающий структурное программирование.

```

main
{
  1: for each node  $N$  in RPO
  2:   vectN <- Placement(N);
  3:   for each predecessor  $P$  of  $N$ 
  4:     vectP <- Assign(P);
  5:     vectN <- Disjunct( vectN, vectP);
  6:     vectP <- Placement(P);
  7:     vectN <- Disjunct( vectN, vectP);
  8:     Set vectN to Placement(N)
  9:   endfor
  10: endfor
} /* main */

```

Сложность алгоритма можно определить как $O(|Edf|)*T()$, где $|Edf|$ – количество дуг DF-графа, а $T()$ – время операции дизъюнкции битовых векторов. Это время согласно оценке (1) можно выразить через множество переменных V как $|V|/size(word)$. Количество дуг DF-графа оценивается как $|N|*|DF|$, где $|DF|$ – множество dominance frontier, соответствующее узлу управляющего графа. Множество $|DF|$ в общем случае

оценивается как $|N|$, хотя данная теоретическая оценка на практике часто вырождается в небольшую константу. Так, для пакета Specint92 статистика по DF приведена в табл. 1. Таким образом, оценка сложности для анализа достижимости в двухфазном алгоритме представляет собой

$$O(|Edf|) * |B|/size(word) \quad (2)$$

На практике она вырождается в

$$O(|N|) * |B|/size(word),$$

так как в реальных программах множества DF содержат очень небольшое количество узлов управляющего графа.

В реализации пакета битовых векторов оптимизирующего компилятора проекта Эльбрус константа $size(word)$ равна 32. Среднее количество объектов, для которых вычисляется множества IDF на задачах Specint92 составляет около 90 (табл. 1). При таких условиях величина $|B|$ перестает влиять на оценку и алгоритм становится линейным по отношению к количеству узлов управляющего графа $O(|N|)$.

3. Групповое построение ϕ -функций в контексте линейного алгоритма

Метод группового построения ϕ -функций может быть использован не только в контексте двухфазного алгоритма, но и в контексте линейных относительно дуг управляющего графа алгоритмах. Рассмотрим алгоритм, предложенный в работе [2]. Это простой однопроходный алгоритм сложностью $O(E)$, решающий задачу нахождения множества IDF для одной переменной. Предлагаемая модификация алгоритма работает как построитель DF-графа на исходном подмножестве узлов управляющего графа. Фактически DF-граф не строится, а осуществляется распространение векторов, аналогично распространению в двухфазном алгоритме. В измененном таким образом алгоритме теоретическая оценка определяется как (2). Преимуществом данного алгоритма является то, что благодаря отсутствию построения DF-графа уменьшается его реализационная сложность.

Далее приведем описание алгоритма. Для полноты описания нам понадобятся несколько определений и утверждений.

Определение 1.

Дуга $x \rightarrow y$ управляющего графа называется *join дугой* (или J-дугой) если x не строго доминирует y .

Определение 2.

DJ-графом называется граф множество узлов которого совпадает с множеством узлов управляющего графа, а множество дуг состоит из объединения дуг дерева доминаторов и J-дуг управляющего графа.

Верна следующая лемма.

Лемма.

Узел z управляющего графа принадлежит множеству $DF(x)$ тогда и только тогда когда существует узел y принадлежащий $SubTree(x)$ с дугой $y \rightarrow z$, которая является J-дугой и $z.level \leftarrow x.level$.

За доказательством и другими подробностями мы отсылаем читателя к работе [2], а у нас теперь есть все необходимое для того, чтобы продолжить изложение основного результата этой работы.

В исходном алгоритме порядок обхода узлов был в соответствии с убыванием уровней дерева доминаторов. В порядке обхода узлов в данной модификации добавляется упорядоченность согласно RPO нумерации в рамках каждого уровня дерева доминаторов, что бы обеспечить обход предшественников узла раньше, чем преемников.

Подробное описание алгоритма приводится в Приложении А. На него мы будем ссылаться в следующих главах данной работы.

4. Доказательство корректности и оценка сложности модифицированного алгоритма

4.1. Доказательство корректности

Для доказательства корректности алгоритма достаточно показать, что в процессе его работы корректно моделируется распространение свойства достижимости на DF графе, аналогично тому, как это происходит для модификаций алгоритмов из [4], представленных в главе 2 этой работы. Таким образом, нужно доказать, что вектора присваиваний распространяются по DF множествам узлов, начиная с исходных, в порядке, когда предшественники DF отношений обходятся раньше, чем преемники. Рассмотрим сначала случай сводимого графа.

Сначала покажем, что в процессе работы алгоритма вектора присваиваний распространяются в DF множества исходных узлов. Процедура `DomFrontVisit` распространяет вектор присваиваний текущего узла (`CurrentRoot`) во все узлы из множества DF этого узла. Это множество формируется из преемников J-дуг поддерева дерева доминаторов данного узла, удовлетворяющих условию леммы 3.1 (строки 4, 19 процедуры `DomFrontVisit`). Для избежания повторного прохода по всему поддереву дерева доминаторов от исходного узла в уже пройденных узлах сохраняются списки их DF множеств. В случае, если в поддереве исходного узла часть узлов была уже пройдена, то обход заканчивается на этих узлах, а кандидаты на попадание во множество DF текущего узла выбираются из DF множества пройденного узла (цикл на строке 18 процедуры `DomFrontVisit`). Только среди этих узлов могут оказаться узлы, у которых уровень в дереве доминаторов будет не больше, чем уровень текущего узла.

В процессе определения множества DF для исходного узла процедурой `DomFrontVisit` новые узлы из этого множества DF (не содержащиеся в множестве `AlfaSet`) заносятся в список исходных узлов `PassNodeList` (строка 13 процедуры `DomFrontVisit`) для последующей обработки. Таким образом, начиная с исходных узлов, в графе формируется подмножество узлов, для которых нужно определять множества DF. Этот процесс можно интерпретировать как построение подграфа DF-графа, необходимого для решения задачи для исходного множества узлов. Итак, мы показали, что процедура `DomFrontVisit` распространяет вектора присваиваний во множества DF для исходного узла `PassNodeList` и в список исходных узлов в результате работы алгоритма попадают все узлы подграфа DF-графа, необходимые для решения задачи. Остается показать, что порядок обработки узлов обеспечивает проход предшественников DF отношений раньше, чем преемников.

Узлы списка `PassNodeList` упорядочены по убыванию уровней узлов в дереве доминаторов и в рамках каждого уровня по возрастанию номеров RPO нумерации, заданной на управляющем графе. Согласно условию леммы преемники DF отношений имеют уровень в дереве доминаторов не больший, чем предшественники, поэтому обход узлов с убыванием уровней дерева доминаторов обеспечивает проход предшественников DF отношений раньше, чем преемников для случая разных уровней

дерева доминаторов. Остается упорядочить обход в пределах одного уровня дерева доминаторов. Этот порядок можно задать RPO нумерацией на управляющем графе. Как известно, RPO нумерация сводимого графа обеспечивает обход предшественников раньше, чем преемников. DF-граф узлов одного уровня дерева доминаторов состоит из дуг, которые полностью соответствуют J-дугам DJ графа, и которые в свою очередь полностью соответствуют дугам управляющего графа. Таким образом, RPO нумерация узлов управляющего графа задает RPO нумерацию на узлах DF-графа, соответствующих одному уровню в дереве доминаторов.

Для случая несводимого цикла DF-граф содержит нетривиальные циклы и для того, чтобы распространение векторов было однопроходным, в алгоритме с явным построением DF графа (глава 2) нужно строить сильно связанные компоненты. В данном алгоритме случай несводимого цикла обрабатывается менее эффективно. Вектора на каждом шаге распространяются не во множество DF, а во множество IDF (строки 7, 22 процедуры DomFrontVisit). Такой подход обеспечивает корректность для случая, когда преемник DF отношения обрабатывается раньше, чем предшественник. Понятно, что проход по всем достижимым в DF графе узлам для каждого исходного узла вносит избыточность в алгоритм, но обеспечивает корректность для случая несводимого графа.

4.2. Оценка сложности

Оценка сложности данного алгоритма полностью аналогична оценке (2). Выигрыш появляется только в реализации, когда задача решается без явного построения DF-графа. Данная модификация алгоритма [2] может быть эффективно использована вместо исходного алгоритма в случае выполнения следующих условий:

1. Управляющий граф является сводимым.
2. $O(|E|*B) \gg O(|N|*|DF|)*B/\text{size}(\text{word})$, т.е. оценка сложности исходного алгоритма много больше оценки сложности модифицированного алгоритма.

Именно такие условия характерны для реальных программ, что подтверждают наши экспериментальные данные (глава 5).

Для несводимого графа в приведенной оценке вместо $|DF|$ присутствует $|IDF|$, что увеличивает сложность реализации алгоритма. Здесь необходимо отметить, что для случая несводимого графа управления данный алгоритм имеет худшую оценку сложности, чем модифицированный двухфазный алгоритм Bilardi и Pingali [4], так как мы не строим DF-граф в явном виде. Это ведет к тому, что в случае несводимого графа моделирование достижимости в контексте данного алгоритма реализуется менее эффективным способом. С другой стороны, в предлагаемом подходе экономится количество проходов.

5. Экспериментальные результаты

Целью эксперимента было сравнение временных характеристик алгоритма из работы [2] и его модификации, рассмотренной в настоящей статье. Сравнение проводилось в контексте работы оптимизирующего компилятора проекта «Эльбрус» [8] на реальных задачах. Для анализа полученных результатов нужно было также собрать статистику по размерам множеств DF и B. Техника эксперимента состояла из последовательного запуска алгоритмов для нахождения множеств IDF во время работы оптимизирующего компилятора. Исходный алгоритм запускался для каждой переменной отдельно, а модифицированный на все множество переменных одновременно. При работе компилятора собирался профиль, на основании которого

были получены временные результаты табл. 1. Эксперимент проводился на следующих задачах пакета SPECint92.

Пакет SPECint92

Ø 008.espresso	задача минимизации булевских функций
Ø 022.li	интерпретатор языка lisp
Ø 023.eqntott	получение таблиц истинности
Ø 026.compress	утилиты сжатия/восстановления данных (Unix)
Ø 072.sc	электронные таблицы
Ø 085.gcc	компилятор с языка C

Результаты эксперимента показывают, что модифицированный алгоритм позволил на два порядка улучшить временные характеристики исходного алгоритма. Такое улучшение объясняет статистика экспериментального пакета. Как мы видим в табл. 1, размер множества DF не больше двух, а среднее количество объектов, на которых работает компилятор, равно 90. Подставляя эти цифры в оценку алгоритма, получаем линейный O(E) алгоритм для множества переменных на реальных задачах.

Табл. 1.

Сводная таблица результатов исследований

Тест	Размер DF для узла средний (макс.)	Размер В для процедуры средний (макс.)	Время работы группового алгоритма (с)	Время работы оригинального алгоритма (с)	Ускорение по времени работы (отношение)
008.espresso	1.27(48)	98(268)	0.38	38.03	100
022.li	1.18(20)	86(132)	0.12	3.03	25
023.eqntott	2.04(21)	91(121)	0.04	6.98	174
026.compress	1.47(7)	93(158)	0.03	2.60	86
072.sc	1.59(151)	95(334)	0.34	42.78	125
085.gcc	2.07(248)	93(438)	5.75	746.92	129

Заключение

В данной работе предложен метод, существенно ускоряющий работу алгоритмов построения ϕ -функций. Использование этого метода расширяет возможности использования SSA-представления в промышленных оптимизирующих компиляторах. С его помощью удалось на два порядка улучшить временные характеристики лучших на данный момент алгоритмов.

Список литературы

1. **Muchnick, Steven S.** Advanced Compiler Design and Implementation – Morgan Kaufman, San Francisco, 1997, chapter 7.2.
2. **Sreedhar, Vugranam C. and Gao, Guang R.** A linear time algorithm for placing ϕ -nodes // Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 1995. Pp. 62-73.
3. **Sreedhar Vugranam C., Lee, Yong-fong, Gao, Guang R.** DJ-Graphs and Their Applications to Flowgraph Analyses – ACAPS Technical Memo 70, May 11, 1994.
4. **Bilardi G., Pingali K.** The Static Single Assignment Form and its Computation – Cornell University Technical Report, July, 1999.

5. **Stoltz, Eric James.** Intermediate Compiler Analysis via reference Chaining – Portland State University, Thesis, January 1995.
6. **Aho, Alfred V., Sethi R., Ullman, Jeffrey D.** Compilers: Principles, Techniques, and Tools – Addison-Wesley, Reading, 1986, chapter 9.4.
7. **Tarjan, Robert E.** Depth first search and linear graph algorithms // SIAM Journal on Computing, 1(2), June 1972. Pp. 146-160.
8. **K. Dieffendorf.** The Russians Are Coming. Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee // Microprocessor Report, V.13, № 2. – February 15, 1999. – P.1-7.

Приложение А.

Описание группового (для множества переменных) алгоритма построения IDF

Используемые обозначения:

DJ-граф $DJ = (N, E)$, где **N** – множество узлов, **E** – множество дуг;

AlfaSet – начальное множество узлов, в которых есть присваивания;

PhiSet – множество узлов, включенных в IDF;

PassNodeList – список обхода узлов. Узлы упорядочены по убыванию уровней узлов в дереве доминаторов и в рамках каждого уровня по возрастанию номеров RPO нумерации, заданной на управляющем графе;

CurrentRoot – текущий узел, для которого вычисляется множество IDF;

Disjunct(*vect1*, *vect2*) – дизъюнкция (операция ИЛИ) двух векторов;

Ниже дано описание алгоритма на алгоритмическом псевдоязыке, легко переводимом в любой современный язык, поддерживающий структурное программирование.

1: Начальная инициализация. Узлы, где есть присваивания, добавляем в список **PassNodeList** в соответствии с нумерацией обхода.

2: Цикл по узлам списка **PassNodeList** с запуском процедуры **DomFrontVisit** построения множества **DF** данного узла и распространения вектора присваивания данного узла в это множество. Перед вызовом процедуры инициализируем переменную **CurrentRoot** текущим узлом. После вызова сохраняем множество **DF** для данного узла цикла.

3: Удаление вспомогательных структур данных.

Процедура **DomFrontVisit**:

1: **Цикл** по всем преемникам узла в **DJ** графе.

2: **Если** преемник соответствует **J** дуге, **то**

- 3: **Если** уровень преемника в дереве доминаторов меньше чем уровень узла **CurrentRoot**, **то**
- 4: Добавляем узел в результирующее **DF** множество узлов процедуры **DomFrontVisit**.
- 5: **Если** узел принадлежит множеству **PhiSet**, **то**
- 6: Обновляем вектор **IDF** преемника операцией **Disjunct** с векторами присваивания и **IDF** текущего узла процедуры **DomFrontVisit**.
- 7: В несводимом графе обновление делаем не только для преемника, но и для текущего множества **IDF** этого преемника. Для этого рекурсивно обходим все сохраненные **DF** множества узлов, начиная с **DF** множества данного преемника.
- 8: **Если** узел не принадлежит множеству **PhiSet**, **то**
- 9: Добавляем узел во множество **PhiSet**.
- 10: Копируем вектора присваивания и **IDF** текущего узла в вектор присваивания преемника.
- 11: **Если** узел не входит во множество **AlfaSet**, **то**
- 12: Добавляем узел во множество **AlfaSet**.
- 13: Добавляем узел в список **PassNodeList**.
- 14: **Если** преемник соответствует **D** дуге, **то**
- 15: **Если** преемник еще не обработан, **то**
- 16: Рекурсивный вызов процедуры **DomFrontVisit** для данного преемника. Вектора присваивания и **IDF** от текущего узла транзитно передаются в вызываемую процедуру и трактуются там как вектора текущего узла вызова.
- 17: **Если** преемник уже обработан, **то**
- 18: **Цикл** по всем узлам из множества **DF** данного преемника.
- 19: **Если** уровень узла в дереве доминаторов меньше чем уровень узла **CurrentRoot**, **то**
- 20: Добавляем узел в результирующее **DF** множество узлов процедуры **DomFrontVisit**.

- 21:** Обновляем вектор **IDF** узла операцией **Disjunct** с векторами присваивания и **IDF** текущего узла процедуры **DomFrontVisit**.
- 22:** В несводимом графе обновление делаем не только для узла, но и для текущего множества **IDF** этого узла. Для этого рекурсивно обходим все сохраненные **DF** множества узлов, начиная с **DF** множества данного узла.
- 23:** В сводимом графе удаляем множество **DF** для данного преемника, так как оно больше не понадобится.

Конец процедуры **DomFrontVisit**.