Глобальный граф потока данных и его роль в проведении оптимизирующих преобразований программ

Дроздов А. Ю., Новиков С. В., Боханко А. С., Галазин А.Б.

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].

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 рисунке 1. приведен пример перевода программы в SSA-форму (слева исходная программа, справа – программа, представленная в SSA-форме).

```
if (...) {
  foo = A;
} else
{
  foo = B;
}
bar = foo;
```

```
if (...) {
  foo1 = A;
} else
{
  foo2 = B;
}
foo3 = φ(foo1, foo2);
bar = foo3;
```

Рис. 1 *Пример перевода программы в SSA-форму*

SSA-форма позволяет в удобном и компактном виде выразить поток данных. Благодаря SSA-форме для любого аргумента, читающего значение некоторой переменной, всегда можно найти одно и только одно определение этой переменной. Определение всегда доминирует использование, то есть все пути программы от ее начала до точки использования переменной проходят через ее определение.

На практике переименование, требуемое канонической SSA-формой, затрудняет распределение переменных на регистры; кроме того, поддержка SSA-формы при проведении каждого оптимизирующего преобразования требует значительных усилий. Однако же главное свойство SSA-формы (возможность для любого аргумента получить доминирующее определение) очень полезно; поэтому были придуманы другие структуры данных, обладающие тем же свойством.

В данной работе описана структура графа определений и использований (Def-Use граф) и методы ее использования в анализе и оптимизациях, которые были реализованы в рамках проекта по созданию оптимизирующего компилятора для микропроцессора Эльбрус-3М [3-5]. Микропроцессор Эльбрус-3М относится к классу EPIC *(Explicitly Parallel Instruction Computing)* архитектур. В архитектурах данного класса активно используется параллельность вычислений на уровне команд для получения высокой производительности вычислительной системы.

1. Def-Use граф

Def-Use граф представляет собой граф потока данных, в котором связь между определениями и использованиями переменных осуществляются способом, который используется при построении формы единственного присваивания. На рис. 2 приведен пример связи записей в переменную A с использованием этой переменной в Def-Use графе. В данном случае определения связываются в точке, являющейся для них итерационным фронтом доминирования, с φ-узлом. В свою очередь φ-узел связывается с использованием переменной.

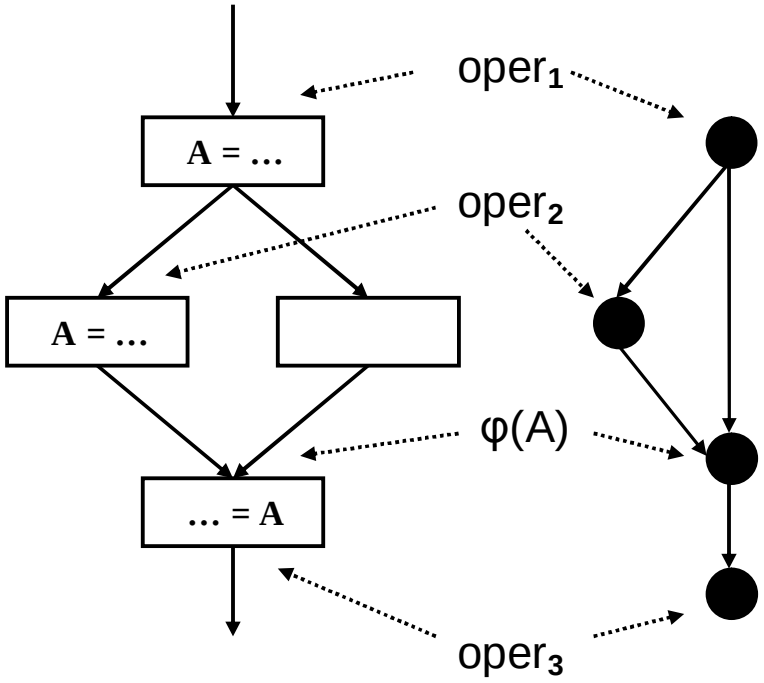

***Рис. 2** Def-Use граф*

Понятие Def-Use графа неразрывно связано с такими понятиями оптимизирующей компиляции как промежуточное представление и граф управления. Опишем эти понятия.

Промежуточное представление – структура данных, фиксирующая состояние(я) программы в процессе компиляции от исходной записи на входном языке до выходного состояния – целевого исходного кода программы, исполняемой на заданной платформе. Основные функции промежуточного представления:

- отображение и сохранение инвариантной семантики исходной программы
- базис для проведения анализа и оптимизирующих преобразований программы
- интерфейс взаимодействия со всеми фазами компиляции, позволяющий фиксировать и передавать изменения программы.

Линейным участком будем называть совокупность операций с выделенными входной и выходной операциями, передача управления в который может происходить только через входную операцию.

Граф управления [1,2] (control flow graph (cfg)) – аналитическая структура, которую логически можно представить как управляющую надстройку над промежуточным представлением. И в реализации наиболее распространенной схемой является представление управляющего графа как отдельного объекта, имеющего взаимно однозначное соответствие с операционной семантикой промежуточного представления, в которой выражена вся полнота семантики передачи управления. Вершинам (узлам) графа управления соответствуют линейные участки, а дугам - управляющие связи между ними, отображающие передачу управления. Через операции начала и конца линейного участка устанавливается двусторонняя связь графа управления с операционной семантикой процедуры. Фактически *cfg* есть факторизация потока управления процедуры, при которой несколько связанных операций, имеющих общую с точки зрения управления судьбу, получают единственного представителя в графе управления.

В EPIC архитектурах поддержан режим предикатного исполнения.

Свойство предикатности - условное исполнение команд, зависящих от значения условного операнда данной команды, называемого предикатом. Когда значение предиката истина команда исполняется нормально; а когда значение предиката ложь, команда игнорируется. Поддержка предикатных вычислений в EPIC архитектурах позволяет выполнять операции раньше перехода, идущего на ее исходный участок, если выполнение операции осуществлять под условием (предикатом) этого перехода. Перенося операцию выше нескольких переходов, предикат операции будет вычислен на основании условий этих переходов. Таким образом, можно называть предикатом операции результат условного выражения, постановка под который позволяет переносить операцию выше заданных переходов.

В оптимизирующем компиляторе в процессе трансляции промежуточное представление модифицируется. Можно выделить этап трансляции, на котором в промежуточном представлении не используется предикатный режим. Этап завершается выполнением фазы, которая использует предикатность для оптимизации программы [6]. После этой фазы промежуточное представление может содержать участки с предикатным кодом. Для указанных этапов трансляции работа с Def-Use графом ведется по-разному. На непредикатном этапе Def-Use граф может быть построен и перестроен в любой момент. На фазе преобразования представления в предикатную форму и на последующих фазах Def-Use граф может быть скорректирован. Построение Def-Use графа на предикатном коде в общем случае аналогично простому связыванию определений и использований и не содержит свойств, используемых анализом и оптимизациями. Поэтому Def-Use граф на предикатном коде не строится, а корректируется с сохранением его аналитических свойств. Описание алгоритма построения Def-Use графа относится к непредикатному представлению программы.

Как видно из примера рис. 2 в Def-Use графе существуют узлы двух типов. Одни узлы соответствуют операциям промежуточного представления, другие являются φ-узлами.

В дугах Def-Use графа и в φ-узлах сохраняются ссылки на переменные, для которых они строятся. Операция может обращаться к некоторому множеству переменных. Из одних переменных операция читает значения, в другие операция записывает результаты вычислений. Поэтому в узел Def-Use графа, который соответствует операции, могут входить дуги для разных переменных.

При построении свойства φ-узлов Def-Use графа полностью аналогичны свойствам φ-функций SSA представления. Все входящие в φ-узел дуги взаимнооднозначно соответствуют входящим в узел управляющего графа дугам. Построенные φ-узлы сохраняются в списке узла управляющего графа. В свою очередь в каждом φ-узле сохраняется ссылка на узел управляющего графа, которому он соответствует.

Для отображения неинициализированных переменных в Def-Use графе создаётся φ-узел неопределенного типа. У этого узла переменная не указывается. На него замыкаются все неинициализированные использования переменных.

2. Алгоритм построения Def-Use графа

Опишем алгоритм построения Def-Use графа (алг. 1).

Алгоритм 1.

1. Определение множества переменных, для которых будет построен Def-Use граф
2. Построение Def-Use графа локально в рамках линейных участков
3. Определение множества переменных, для которых может потребоваться построение φ-узлов
4. Вычисление итерационного фронта доминирования для множества переменных, определенного на шаге 3
5. Построение φ-узлов по результатам шага 4
6. Связывание операций и φ-узлов дугами

На первом шаге алгоритма определяется множество переменных, для которых граф будет построен. Для корректного использования Def-Use графа необходимо, что бы в нем связывались все определения и использования переменной. Это накладывает ограничение на переменные, для которых граф можно построить. Если на переменную был взят адрес, то статическим анализом не всегда возможно определить, где в переменную происходит запись и где переменная читается. Таким образом, Def-Use граф строиться для переменных, про которые известны все места обращения к ним. В

первом приближении можно включать в это множество все скалярные переменные, на которые не берется адрес. Для структурных переменных связь в Def-Use графе отражает поток данных консервативно. Для таких объектов может существовать связь между записью в один элемент и чтением из другого элемента этого объекта. Такой консерватизм не всегда годиться для потокового анализа, зато может быть использован для оптимизации удаления избыточных вычислений. Переменные могут быть как локальными, так и глобальными. Для глобальных переменных усложняется проверка на обращение к ним в местах вызовов процедур. Ответ на вопрос об обращении к глобальным переменным в некотором вызове может быть получен на основании межпроцедурного анализа, либо должен быть дан консервативный ответ.

Второй шаг алгоритма состоит в локальном построении Def-Use графа в рамках линейных участков. Техника построения локального потокового графа намного проще, чем техника построения глобального графа. Эта оптимизация позволяет обработать более простой техникой подмножество локальных переменных, то есть переменных, обращение к которым происходит только в пределах одного линейного участка. Локальное построение потокового графа состоит в построении дуг между ближайшими записями в переменные и их использованиями. Алгоритм имеет линейную сложность и состоит в обходе линейного участка и обработке каждой операции. Для аргументов операции в таблице определений находятся ближайшие записи, а по результатам операций они добавляются в таблицу определений.

На шаге 3 алгоритма формируется множество переменных, для которых может потребоваться построение φ-узлов. В это множество включаются все переменные, для которых не был построен локальный Def-Use граф.

На шаге 4 происходит вычисление итерационного фронта доминирования (iterated dominance frontier (IDF)) для множества объектов, определенного на шаге 3. Для этой цели используется алгоритм группового построения IDF, предложенный в работе [7]. Алгоритм имеет линейную сложность для реальных приложений.

По результатам шага 4 на шаге 5 происходит построение φ-узлов. Эти узлы заносятся в списки узлов управляющего графа. Их можно трактовать как записи в соответствующие им переменные, находящиеся в начале линейных участков.

Окончательное связывание узлов Def-Use графа происходит на шаге 6. Эта задача решается линейным обходом по дереву доминаторов с распространение таблицы определений по дугам управляющего графа [8].

Таким образом, Def-Use граф может быть построен линейным алгоритмом. Такая сложность позволяет регулярно перестраивать Def-Use граф на не предикатном промежуточном представлении, если оптимизация приводит его в некорректное состояние.

3. Использование Def-Use графа в анализе

В настоящем разделе мы приведем пример применения Def-Use графа при анализе потока данных, известного как “нумерация значений” (Value Numbering).

Задачей анализа, известного как “нумерация значений” (Value Numbering) является нумерация всех операций и φ-узлов программы таким образом, чтобы операциям и φ-узлам, вырабатывающим одинаковый результат, соответствовал один номер. Такой номер носит название *класса конгруэнтности*.

Принадлежность нескольких операций и φ-узлов одному классу конгруэнтности дает возможность для применения таких оптимизаций, как Сбор общих подвыражений (Common Subexpression Elimination), Глобальная пропагация копий (Global Copy Propagation), Частичное удаление избыточности (Partial Redundancy Elimination).

Основной структурой данных, на базе которой проводится Value Numbering, как раз и является Def-Use граф. Действительно, для доказательства эквивалентности двух операций необходимо доказать, что их соответствующие аргументы также эквивалентны. В случае константных аргументов такое доказательство тривиально; но в случае переменных аргументов доказательство требует дополнительного анализа.

Если дуги Def-Use графа, соответствующие аргументам анализируемых операций, доступны, алгоритм проведения такого анализа очевиден: достаточно проверить на равенство классы конгруэнтности Def-Use предшественников аргументов. Требуется лишь гарантировать, что предшественники всех Def-Use дуг будут обработаны при Value Numbering'е раньше их преемников. В случае обхода операций и φ-узлов в порядке RPO-нумерации, такая гарантия обеспечивается для всех дуг, кроме тех, что соответствуют обратным дугам управляющего графа. Наличие последних обуславливает итеративную природу алгоритма Value Numbering'а. Кроме того, можно сделать следующее наблюдение: повторную итерацию можно делать не для всех операций и φ-узлов программы, а только для тех из них, что входят в текущую сильно-связную компоненту Def-Use графа. Таким образом, можно значительно ускорить время проведения Value Numbering'а ([9]).

4. Использование Def-Use графа в оптимизациях

Def-Use граф активно используется не только в фазах анализа, но и в оптимизирующих фазах для быстрого получения аналитической информации о потоке данных.

Использование Def-Use графа позволяет за линейное время определять избыточные вычисления на фазе их удаления. Алгоритм размечает вычисления, которые не являются избыточными, обходом Def-Use графа, начинающимся в операциях, которые не избыточны априори.

Другим примером использования Def-Use графа является переименование переменных для повышения параллельности на уровне операций (рис. 3). Возможности для переименования возникают как в исходном коде, так и после оптимизирующих преобразований. В Def-Use графе находятся подграфы для переменных, которые могут быть переименованы в другие переменные.

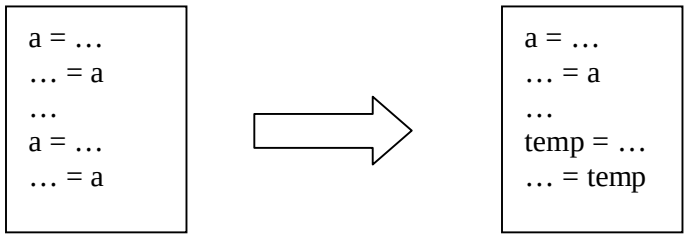

Рис 3. *Переименование переменных*

Еще одной оптимизацией, активно использующей Def-Use граф для анализа, является вынос инвариантных вычислений из цикла. Определить инвариантность можно разметкой операций цикла, достижимых от φ-узлов головы. Достижимость

означает, что какой-то аргумент операции определяется в цикле и такие операции инвариантными не являются. Такой анализ достаточен, если все переменные операции включены во множество для построения Def-Use графа и операция не имеет побочных эффектов. Например, для операций вызова, записи в память или чтения из памяти информации об аргументах и результатах недостаточно, и может потребоваться дополнительный анализ конфликтов с другими операциями цикла.

5. Использование Def-Use графа в распределителе регистров

Задачей распределения регистров ([2]) является распределение произвольного числа пользовательских переменных на строго ограниченное число физических регистров. При этом две переменные не могут быть распределены на один и тот же физический регистр, если “времена жизни” этих переменных пересекаются. Под “временем жизни” переменной понимается участок программы, на котором значение переменной может быть прочитано.

Очевидно, что если значение некоторой переменной `V`, записанное операцией `o1`, может быть прочитано как первый аргумент операции `o2`, то результат `o1` и первый аргумент `o2` должны быть распределены на один и тот же физический регистр.

Можно было бы просто для всех операндов, соответствующих одной переменной, использовать один физический регистр; но такой наивный подход накладывает значительные ограничения на распределитель регистров. Дело в том, что часто одна и та же переменная используется в независящих друг от друга частях программы:

```
for ( i = 0; i < N; i++ ) {
    ...
}
...
for ( i = 0; i < M; i++ ) {
    ...
}
```

В приведенном примере переменная `i` используется в качестве счетчика для двух разных циклов. На самом деле эти два использования `i` совершенно не зависят друг от друга. В случае распределения всех операндов, соответствующих переменной `i`, на один регистр, распределителю регистров пришлось бы задействовать один и тот же номер физического регистра. Это в свою очередь могло бы привести к неоправданному решению о нехватке регистров. Например, если и в первом и во втором цикле есть по одному свободному физическому регистру, но эти регистры имеют разные номера.

Def-Use граф позволяет решить указанную проблему. С его помощью можно отказаться от привязки к пользовательским переменным, и распределять уже “сети” (webs), являющиеся связными компонентами Def-Use графа ([10]). В приведенном примере переменная `i` образует две сети, для которых можно использовать разные физические регистры.

6. Алгоритмы коррекции Def-Use графа.

На этапе перехода представления в предикатную форму и последующих оптимизирующих преобразования требуется коррекция Def-Use графа для сохранения неконсервативных свойств этой структуры данных. Ниже приведены алгоритмы коррекции для различных преобразований управляющего графа и возможных

переносов операций при планировании. Описание алгоритмов дано на алгоритмическом псевдоязыке, легко переводимом в любой современный язык, поддерживающий структурное программирование.

6.1. Коррекция при преобразовании управляющего графа

В данной главе будут рассмотрены такие преобразования управляющего графа как

- вставления пустого узла управляющего графа с перенаправлением на него подмножества дуг (алг. 2);
- объединение узлов ациклического участка в один узел (алг. 3).

Алгоритм 2.

1. Цикл по всем *j* -узлам преемника вставленного узла

2. **Если** на новый узел было перенаправлено как минимум две дуги

3. Создание нового *j* -узла во вставляемом узле управляющего графа

4. Перенаправляем дуги Def-Use графа, ссылающиеся на перенесённые управляющие дуги, на новый *j* -узел

5. **Если** на новый узел была перенаправлена одна дуга управляющего графа

6. Находим дугу Def-Use графа, ссылающуюся на управляющую дугу, ставшую после преобразования единственной входной дугой вставленного узла и корректируем эту ссылку

7. Конец цикла

8. Если после перенесения дуг в преемник вставленного узла входит только одна управляющая дуга

9. Удаление всех *j* -узлов преемника

Алгоритм коррекции Def-Use графа после вставления пустого узла управляющего графа с перенесением на него подмножества дуг имеет линейную сложность относительно φ-узлов преемника вставляемого узла управляющего графа. Данное преобразование используется на этапе работы алгоритма глобального планирования для удаления критических дуг, создания предциклов, постциклов, и т. п.

Другой алгоритм коррекции используется в момент объединения узлов ациклического участка в один узел. Такое преобразование происходит при переходе к предикатному представлению. Поток управления преобразуется в поток данных.

Алгоритм 3.

1. **Цикл** по всем внутренним узлам ациклического участка
2. **Цикл** по всем *j* -узлам текущего узла ациклического участка
3. Коррекция ссылок на узлы и дуги управляющего графа в *j* -узле и входящих в него дугах Def-Use графа
4. **Конец цикла**
5. Удаление списка (но не самих *j* -узлов) в текущем узле управляющего графа
6. **Конец цикла**

Ациклические регионы, преобразуемые в узлы, имеют в своей структуре внутренние и внешние узлы управляющего графа. Внутренними называются узлы, предшественники которых в управляющем графе тоже принадлежат этому ациклическому региону. Внешним можно называть узлы, предшественники которых находятся за пределами ациклического участка. Для предикатных преобразований используются ациклические участки с одним внешним узлом – головой. При преобразовании внутренние узлы удаляются, а операции переносятся в голову. Таким образом, коррекция состоит в том, что бы скорректировать ссылки на управляющий граф в *j* -узлах внутренних узлов управляющего графа. Коррекция ссылок состоит в их обнулении, так как после предикатного преобразования структуры управляющего графа удаляются и само управление преобразуется в поток данных одного участка. Сложность алгоритма линейна относительно φ-узлов внутренних узлов ациклических участков управляющего графа.

6.2. Коррекция при переносе операций

Алгоритмы планирования играют ключевую роль для получения высокой производительности в вычислительных системах, основанных на EPIC архитектурах. Особенно актуальны алгоритмы глобального планирования, осуществляющие перенос операций между линейными участками управляющего графа. Примером такого планирования является Wavefront Scheduling [11]. Этот метод используется в компиляторе фирмы Intel для EPIC архитектуры IA-64 (Itanium 2). Ниже приведены алгоритмы, позволяющие корректировать Def-Use граф для предложенного подхода планирования. При планировании во фронт возможно два случая:

- планирование одной копии исходной операции (алг. 4);
- планирование нескольких копий исходных операций (алг. 5).

В предложенных алгоритмах коррекции используется понятие достижимости. Достижимость по управлению между двумя узлами означает наличие пути в управляющем графе между ними. Информация эта сохраняется таблично и ответ на вопрос о достижимости осуществляется за константное время.

Алгоритм 4.

1. **Цикл** по аргументам копии исходной операции, для которых построен Def-Use граф

2. **Если** предшественник в Def-Use графе исходной операции для соответствующего аргумента является операцией

3. Построение дуги между предшественником и копией для текущего аргумента

4. **Если** предшественник в Def-Use графе исходной операции для соответствующего аргумента является j-узлом и доминирует копию

5. Построение дуги между предшественником и копией для текущего аргумента

7. **Если** предшественник в Def-Use графе исходной операции для соответствующего аргумента является j-узлом и не доминирует копию

8. Поиск предшественников этого j-узла, от которых достижима копия операции

9. **Если** найден один предшественник

10. Построение дуги между предшественником и копией для текущего аргумента

11. **Если** найдено несколько предшественников

12. Построение j-узла, не привязанного к узлам управляющего графа, для всех предшественников и дуга Def-Use графа строится между данным узлом и копией операции.

13. **Конец цикла**

14. Перенос преемников операции в Def-Use графе на копию

В цикле 1-13 алгоритма 4 осуществляется коррекция Def-Use графа для аргументов переносимой операции. Идея коррекции состоит в поиске доминирующего набора определений копии переносимой операции. Этот поиск осуществляется по Def-Use графу и в худшем случая является линейным по количеству узлов Def-Use графа. На практике сложность существенно меньше, так как обход осуществляется только для одной переменной и размер подграфа одной переменной по отношению к размеру всего Def-Use графа невелик.

Коррекция Def-Use графа для результатов операции осуществляется на шаге 14 и состоит в переносе преемников исходной операции на копию.

При переносе операций с построением нескольких копий существенно усложняется коррекция Def-Use графа для результатов копий операции (алг. 5), так как дублирование определений приводит к необходимости построения *j* -узлов, привязанных к управлению от точек вставки копий операции до точки, в которой стоит исходная операция. Для этого задаётся список определений (копий операции), список использований (использований операции), точка от которой надо начать строить *j* -узлы, маркер зоны, в которой можно их строить. Разметка зоны для построения неконсервативных с точки зрения привязки к управляющему графу φ-узлов, необходима для случая условного переноса операций. Маркером размечается зона, где происходит безусловный перенос. Дело в том, что для смешанного вида промежуточного представления (когда одновременно существует и предикатный код, и поток управления) не консервативно удается скорректировать Def-Use граф только для операций, переносимых безусловно. Этот консерватизм ослабляет возможности для потокового анализа, но не для распределения регистров. Алгоритм представляет собой обход управляющего графа от текущей точки вверх до определений. Определения должны групповым образом доминировать точку начала обхода, то есть все пути от начала программы к точке начала обхода должны проходить хотя бы через одно определение.

Алгоритм 5.

рекурсивный_обход_подграфа_управления( )

{

1. **Если** текущий узел управляющего графа не входит в зону построения *j* -узлов

2. Построение *j* -узла, не привязанного к узлам управляющего графа, для всех определений. Все использования связываются с ним

3. **Если** для текущего узла уже построен *j* -узел

4. Остановка обхода; в качестве определения используется этот *j* -узел. Все использования связываются с ним

5. Определения списка разбиваются по принадлежности текущему узлу.

6. **Если** есть определения в текущем узле

7. Построение *j* -узла, не привязанного к узлам управляющего графа, для этих определений и принятие его за определение, которое будет соответствовать входу в узел. Все использования связываются с построенным

```
        j -узлом

8.      Построенный j -узел добавляется    в список
        использований вместо  остальных использований.

9.  Если список внешних определений пуст, обход
    прекращается.

10. Если список внешних определений одноэлементный

11.     Единственное определение связывается с
        использованиями и обход прекращается

12. Если в текущий узел входит одна дуга

13.     рекурсивный_обход_подграфа_управления( ) для
        предшественника дуги

14. Построение j -узла в текущем узле управляющего графа.
    Для каждой входящей дуги определяется собственный
    подсписок определений на основании информации о
    достижимости. j -узел связывается со всеми
    использованиями. Для каждого предшественника
    дуги вызывается рекурсивно процедура
    рекурсивный_обход_подграфа_управления() с
    собственным подсписком определений и
    списком использований, состоящим из построенного
    j -узла
} /* рекурсивный_обход_подграфа_управления */
```

Сложность предложенного алгоритма линейна по количеству узлов управляющего графа, находящихся между набором узлов, содержащих определения, и исходной точкой обхода. Таким образом, худшая оценка алгоритма определяется размерами планируемых регионов. Согласно статистике, приведенной в работе [11], регионы глобального планирования в 60% случаев содержат менее 5 линейных участков. Процент регионов, содержащих более 20 линейных участков крайне мал. В связи с этим, алгоритм может быть использован для коррекции Def-Use графа без заметных временных издержек.

Заключение

В данной работе описана важная для оптимизирующей технологии структура данных - Def-Use граф. В работе рассмотрены алгоритмические аспекты построения данной структуры, ее использования в анализе и оптимизациях, а также коррекции при оптимизирующих преобразованиях. В ней также показано, что использование Def-Use графа позволяет эффективно решать многие задачи оптимизирующей компиляции, такие как потоковый анализ, скалярные оптимизации и распределение регистров. Благодаря практически линейной сложности построения и коррекции этой структуры данных, является возможным ее многократное использование практически на всех фазах оптимизирующей трансляции. Описанные в работе методы применяются в

контексте оптимизирующего компилятора, который используется как для EPIC архитектур Эльбрус-3М и Itanium 2, так и для суперскалярной архитектуры Sparc.

Список литературы

1. **Muchnick, Steven S.** Advanced Compiler Design and Implementation – Morgan Kauffman, San Francisco, 1997, chapter 7.2.
2. **Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman**, “Compilers: principles, techniques, and tools”, Addison-Wesley, Reading, Massachusetts, 1986
3. **Boris Babayan**. E2K Technology and Implementation. // in Proceedings of the Euro-Par 2000 - Parallel Processing: 6th International. - Volume 1900 / 2000. – January, 2000. – P. 18-21.
4. **М. Кузьминский**. Отечественные микропроцессоры: Elbrus E2K // Открытые системы, № 05-06, 1999. – С. 8-13.
5. **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.
6. **Дроздов А. Ю., Новиков С. В., Шилов В. В.** “Эффективный алгоритм преобразования потока управления в поток данных”, Информационные технологии. № 2. 2005. Приложение. С. 24-31.
7. **Дроздов А. Ю., Новиков С. В., “**Эффективный алгоритм построения формы статического единственного присваивания”, Информационные технологии. № 3. 2005.
8. **Vugranam C. Sreedhar, Yong-fong Lee, Guang R.Gao**. DJ-Graphs and Their Applications to Flowgraph Analyses. ACAPS Tchnical Memo 70, May 11, 1994.
9. **Loren Taylor Simpson,** “Value-Driven Redundancy Elimination”, Ph.D. Thesis, Rise University, Houston, Texas, 1996
10. **Gregory Chaitin, Mark Auslander, Ashok Chandra, John Cocke, Martin Hopkins, Peter Markstein,** “Register allocation via coloring”, Computer Languages, January 1981, pp. 47-57
11. **Jay Bharadwaj, Kishore Menezes, Chris McKinsey,** “Wavefront scheduling: path based data representation and scheduling of subgraphs”, Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture, pp. 262-271, 1999