ООО "ИНТЕРСОЦИОИНФОРМ"
***** КОМПЬЮТЕРЫ В УЧЕБНОМ ПРОЦЕССЕ *** № 5, май 2005 *****
--------- СТРАННЫЙ СПЕЦИАЛЬНЫЙ ЖУРНАЛ ДЛЯ ПЫТЛИВЫХ УМОВ ЛЮБОГО ВОЗРАСТА---------

АНАЛИЗ “МЕЖПРОЦЕДУРНАЯ НУМЕРАЦИЯ ЗНАЧЕНИЙ”

А.Ю.Дроздов,
А.В.Кан
Институт микропроцессорных вычислительных систем РАН, Москва
E-mail: sasha@mcst.ru; akan@mcst.ru

1. ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ

Компилятор – это программа, которая переводит программу, представленную на одном языке, на семантически эквивалентную программу на другом языке. В процессе такого преобразования компилятором, как правило, используются одно или несколько промежуточных представлений программы [1]. Именно над промежуточным представлением производятся различные виды анализов и оптимизаций. В статье будет описано промежуточное представление, используемое для анализа “Межпроцедурная нумерация значений”.

За основу промежуточного представления взято представление EIR, которое применяется в языковом оптимизирующем компиляторе ecf_opt, разрабатываемом компанией Elbrus Inc. (Россия) [2, 3].

EIR (Elbrus Intermediate Representation) является высокоуровневым машинно-независимым промежуточным представлением семантики, предназначенным для представления семантики ряда входных универсальных языков высокого уровня с возможностью последующей генерации машинно-зависимых представлений ряда платформ.

1.1. Управляющий граф

Управляющий граф процедуры является аналитической структурой данных, отражением результатов анализа топологии и семантики программы. Каждый узел управляющего графа соответствует некоторому линейному участку. Управляющий граф является ориентированным. Каждая дуга такого графа соответствует возможности передачи управления в программе между линейными участками.

Линейным участком называется упорядоченное множество операций. Если множество не пусто, то выделяются входная и выходная операции и выдвигаются требования, которым должен удовлетворять линейный участок:

– *Связность*: все операции линейного участка, включая конечную, достижимы из начальной операции без выхода за пределы линейного участка; из всех операций линейного участка, включая начальную достижима конечная операция.

– *Замкнутость*: выход из линейного участка минуя конечную операцию или вход в линейный участок минуя начальную операцию невозможен.

– *Полнота*: линейному участку принадлежат все операции проходимые при обходе сверху от начальной операции к конечной и при обходе снизу от конечной операции к начальной.

– *Одноуровневость*: линейный участок не содержит внутри себя других линейных участков.

– *Однозначность*: каждой операции множества конечных операций взаимно однозначно соответствует операция множества начальных операций и наоборот; линейный участок, которому принадлежит операция, определяется однозначно обходом в любом направлении.

– *Ацикличность*: ни одна операция линейного участка не может быть достигнута при проходе по управлению от самой себя без выхода за пределы линейного участка.

– В линейном участке может быть не более одной операции записи.

Все узлы графа делятся на два типа: обычные узлы и узлы слияния. Обычный узел может иметь не более одного предшественника. Узлы слияния могут иметь произвольное число предшественников. Линейные участки, соответствующие узлам слияния не содержат операций, кроме псевдоопераций JOIN.

Один узел графа помечен как стартовый, такой узел не имеет входных дуг. Стартовый узел соответствует линейному участку, с которого начинается выполнение процедуры. Один узел графа помечен как стоповый, в него сведены все дуги, по которым может происходить выход из процедуры.

Граф может содержать циклы. Представим, что мы обходим граф “вширь” помечая уже рассмотренные узлы. В процессе такого обхода могут встретиться дуги, ведущие к уже помеченному узлу. Такие дуги будем называть *обратными*.

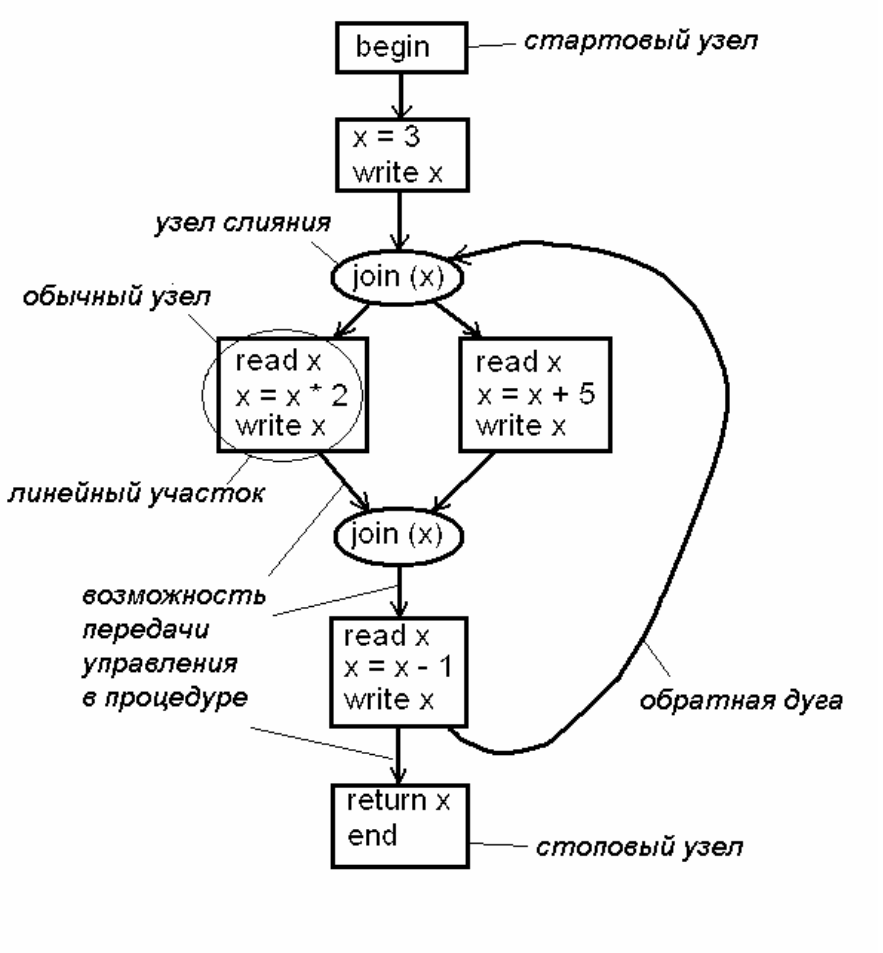

Рис. 1. Управляющий граф.

Для дальнейшего изложения необходимо ввести понятие множество IDF (immediate dominant frontiers) для узла. IDF для узла *x* – это множество ближайших узлов слияния, встречающихся на всех путях от узла *x* до стопового узла.

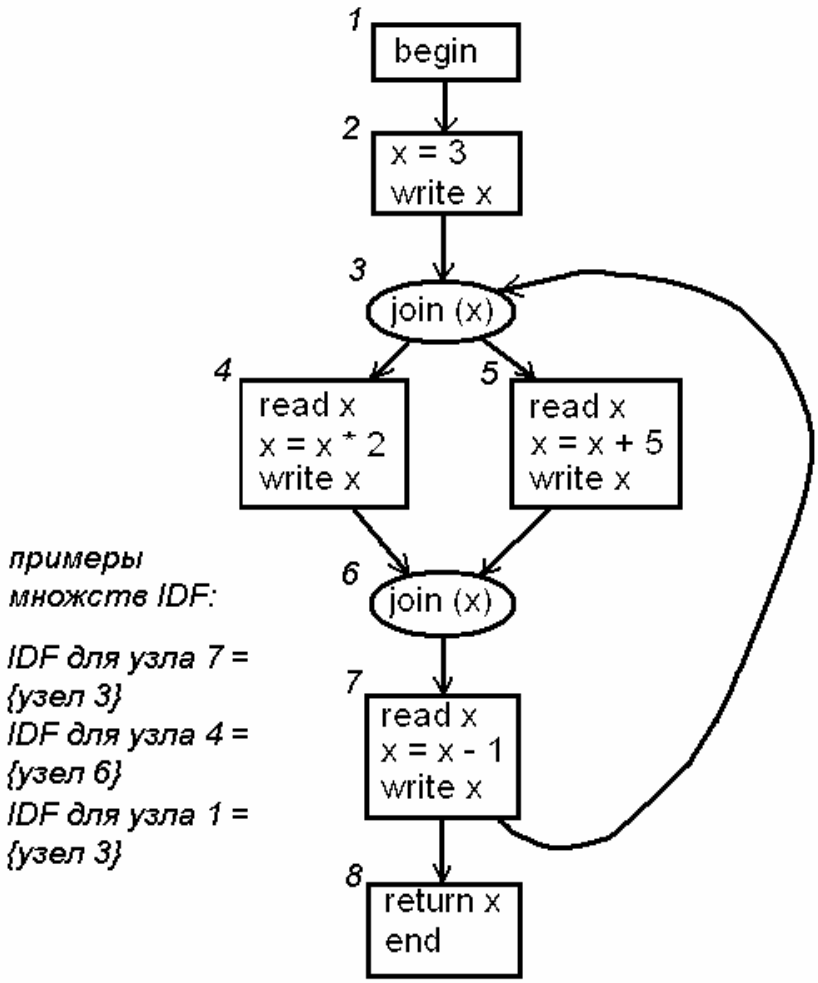

Рис. 2. Множества IDF.

1.2. Операционная семантика

Операции делятся на две группы – те, которые вырабатывают результат (результативные), и те, которые результата не вырабатывают. Будем считать, что операция может вырабатывать не более одного результата. Операции будут записываться в виде:

OP_NAME R_{i1} R_{i2} ... $R_{in} \rightarrow R_{im}$,

где R_{i1} R_{i2} ... R_{in} – регистры-аргументы операции, R_{im} – результат.

Имя операции имеет контекстно-независимую семантику: при любых условиях, если операции с одним именем подавать в соответствующие аргументы одинаковые значения, будет получаться одинаковое значение результата.

Вызов процедуры осуществляется операцией **CALL**.

CALL R_{i0} R_{i1} R_{i2} ... $R_{in} \rightarrow (R_{im})$,

где R_{i0} – адрес вызываемой процедуры, либо символьное имя вызываемой процедуры; R_{i1} R_{i2} ... R_{in} – регистры-аргументы операции – параметры, передающиеся в вызываемую функцию; R_{im} – возвращаемое процедурой значение.

Имя операции **CALL** имеет контекстно-независимую семантику; тем не менее, если операции с одним именем подавать в соответствующие аргументы одинаковые значения, возможны различные значения результата.

Работа с памятью осуществляется посредством чтения/записи объектов. Объект – это непрерывная последовательность N байт в памяти, причём если объект записан начиная с адреса X, то в диапазон адресов [X, X+N–1] другие объекты не попадают. Таки образом, вся работа с памятью осуществляется через операции **READ** $x \rightarrow R_{i1}$ и **WRITE** $R_{i1} \rightarrow x$ (читать значение объекта *x* в регистр R_{i1} и записать значение в регистре R_{i1} в объект *x*). Объекты могут быть глобальными и локальными (иметь признак глобальности/не иметь этого признака). Глобальный объект доступен из любой процедуры программы. Локальный объект доступен только из процедуры, в которой он описан.

код на Си	*промежуточное представление*
...	...
x = 1;	MOV 0X1 -> R1
y = 2;	MOV 0X2 -> R2
z = x*(--x) + y*(--y);	SUB R1 0X1 -> R3
printf("%d\n", z);	SUB R2 0X1 -> R4
...	MUL R1 R3 -> R1
	MUL R2 R4 -> R2
	ADD R1 R2 -> R1
	MUV [symtbl] -> R5
	CALL <printf> R5 R1
	...

Рис. 3. Код на Си – промежуточное представление.

Итак, в рамках анализа нас будут интересовать только результативные операции, которые, в свою очередь, могут быть четырех типов: **READ**, **WRITE**, **CALL** и др.

Повторим, что описанное промежуточное представление является необходимым для проведения анализа “Межпроцедурная нумерация значений” алгоритмом, приведенным в настоящей работе. Вместе с тем, оно является достаточно общим, – так, всегда (почти всегда) удаётся отобразить любое используемое промежуточное представление к вышеуказанному, без потери семантики программы.

2. НУМЕРАЦИЯ ЗНАЧЕНИЙ

2.1. Основные понятия

Нумерация значений есть технология, суть которой заключается в следующем: каждому значению, вычисляемому в программе, ставится в соответствие число, называемое “номер значения”, таким образом, что два значения получают один номер, если компилятор может доказать, что эти значения равны для всех возможных входных данных в программу [4].

Номера присваиваются значениям, но значения в программе вырабатываются операциями. Таким образом, можно говорить о присвоении номеров значений операциям (для результативных операций). Операции, вырабатывающие одинаковые значения, называются эквивалентными. Такие операции получат один номер значения. Всё множество результативных операций, разбивается на подмножества – классы эквивалентности (конгруэнтности).

Номера значений будут представляться целыми числами, начиная с UNDEF_VAL=–1. Номер UNDEF_VAL=–1 будет обозначать неопределённый номер, когда ничего нельзя сказать о значении, которое вырабатывает операция. Две операции с одним и тем же номером, равным UNDEF_VAL, не являются эквивалентными.

Функция int GetNewValNum(void) будет вырабатывать номер значения, который ещё не был использован.

Сформулируем задачу анализа: имеется управляющий граф процедуры (см. раздел 1), требуется присвоить операциям номера значений, в рамках ограничений, указанных в предыдущем абзаце.

Известны различные алгоритмы для нумерации значений в процедуре. Опишем алгоритм, использующий хеширование операций.

2.2. Моделирование работа с памятью

Работа с памятью на промежуточном представлении была описана в разделе 1.2 (через записи/чтения объекта). Для нужд анализа будем поддерживать структуру, моделирующую работу с памятью. Она позволит запоминать *когда* (в каком узле), *куда* (в какой объект) и *что* (какой номер значения) было записано. Соответственно, когда встретится операция чтения, можно будет узнать, какое значение было записано, в ближайшем узле записи, доминирующим узел чтения.

Для работы с такой моделью будут использоваться функции:

void SetObjValNum(node, obj, val_num) – запоминание того факта, что в узле node происходит запись в объект obj, значения, которое имеет номер значения val_num.

int GetObjValNum(node, obj) – получает номер значения, которое было записано в объект в ближайшем узле, доминирующем узел node.

Если происходит чтение объекта, в который не было записи, то GetObjValNum возвращает UNDEF_VAL.

Проблема возникает в местах слияния ветвей исполнения программы. Представим, что в узле 1 мы пишем в объект *x* значение A, в узле 2 пишем в объект *x* значение B, сливаем управление от узлов 1 и 2 в узел 3 (узел слияния), а в узле 4 происходит чтение *x*. Какую запись считать доминирующей?

Решение проблемы заключается в том, что в узлах слияния мы будем имитировать запись (вызовом SetObjValNum) для каждого объекта, по которому происходит слияние. Таким образом, ближайшей доминирующей записью для чтения в узле 4, будет запись в узле 3. Заметим, что так мы всегда будем получать единственную ближайшую доминирующую запись, поскольку все слияния упираются в узлы слияния, а в этих узлах строятся записи.

Рассмотрим пример (рис. 4). В узле слияния будем имитироваться запись в объект *x*. Но какое значение будет записано: A или B? В таких случаях будет применяться принцип “слияния номеров значений”: если оба значения, имеют один номер, то этот номер и

будет результатом слияния; иначе, результатом слияния будет номер UNDEF_VAL. Этот принцип реализован в функции

int JoinValNums(val_num1, val_num2);

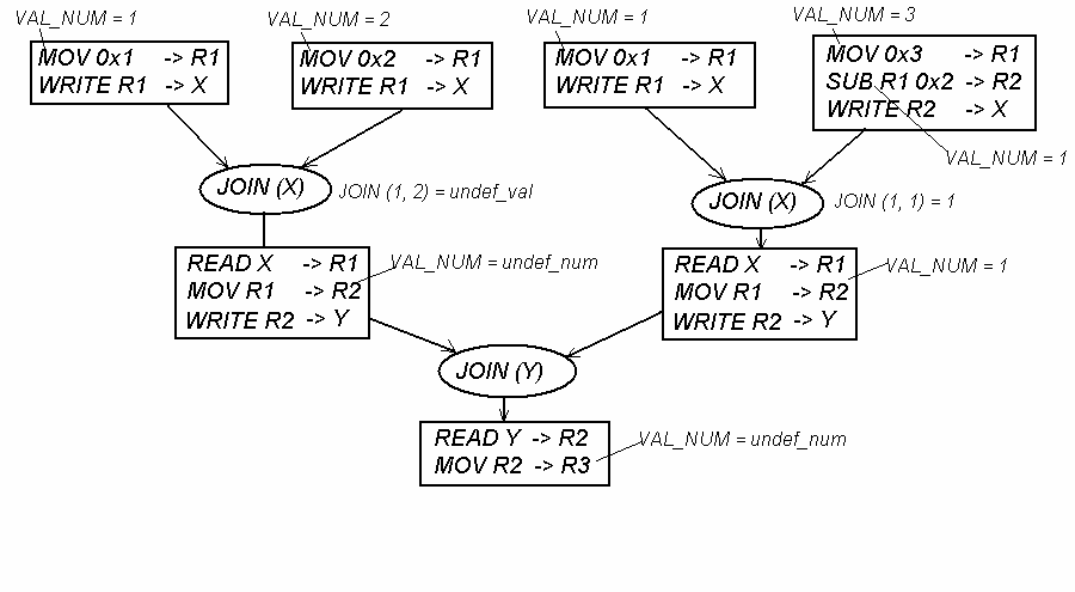

Рис. 4. Слияние значений.

2.3. Описание алгоритма

Представленный здесь алгоритм проведения анализа нумерации значений является одной из реализаций техники проведения анализа с использованием хеширования операций, предложенной в [4] (Hash Based Value Numbering).

Обрабатываем узлы управляющего графа процедуры в таком порядке, при котором, узел может быть обработан только тогда, когда обработаны все его предшественники, за исключением предшественников по обратным дугам. Каждый узел может быть либо узлом слияния, либо обычным узлом.

Рассмотрим сначала обработку обычного узла. Проходим последовательно, начиная с первой, по всем операциям узла. До раздела 3 исключим из рассмотрения операцию **CALL**. Нерезультативные операции также пропускаем.

Будем использовать две хеш-таблицы. Первая из них – ConstHash, хеш-таблица констант. Каждая запись имеет ключ и данные (в качестве данных выступает номер значения). Ключ – это константа, которую вырабатывает операция. Вторая хеш-таблица, OpersHash – это хеш-таблица операций. Каждая запись имеет составной ключ и данные, в качестве которых выступает номер значения. Составной ключ – имя операции и номера значений всех её аргументов.

Для операций, вырабатывающих константное значение, (например **MOV** 0x0 → R1; **ADD** 0x1 0x2 → R2) нужно лишь запоминать в хеш-таблице, какие значения констант уже встречались и какие номера им были назначены. Если константа ещё не встречалась, назначаем ей любой ранее не использованный номер значения. В любом случаем полученный номер значения присваиваем операции.

Для операций чтения просто получаем номер значения, которое было записано в объект ближайшей доминирующей записью. Этот номер значения присваиваем операции.

```
EvalConstOper( oper)
{
   /* получаем константу – результат операции */
   const = GetOperConstResult( oper);
   /* ищем номер значения в таблице */
   if ( val_num is not found in ConstHash by const )
   {
      /* получаем ещё не использованный номер */
      val_num = GetNewValNum();
      /* создаём новую запись */
      CreateEntryWithValNumByConst( ConstHash, val_num, const);
   }
}

EvalReadOper( oper)
{
   /* получаем объект, который читает операция READ */
   obj = GetOperReadObj( oper);
   /**
    * получаем номер значения, которое находится в объекте
     * (было в объект записано) к моменту исполнения узла node
     */
   val_num = GetObjValNum( node, obj);
}
```

Для операций записи получаем val_num = номер значения регистра-аргумента, которое мы хотим записать, и запоминаем тот факт, что в текущем узле производится запись в объект значения с номером val_num. Номер значения присваиваем операции.

Кроме того, во множество объектов каждого узла слияния из множества IDF для текущего узла нужно добавить объект (в описании обработки узлов слияния будет показано, для чего это нужно).

```
EvalWriteOper( oper)
{
   /* получаем объект, в который пишет операция WRITE */
   obj = GetOperWriteObj( oper);
   /* получаем номер значения, которое пишет операция WRITE */
   val_num = GetNthArgValNum( oper, 1);
   /* запоминаем, что была запись в узле node */
   SetObjValNum( node, obj, val_num);
   /* "проталкиваем" объект дальше по всем IDF узла */
   PushObjToAllIDFs( node, obj);
}
```

Для прочих результативных операций справедливо: если у двух операций с одинаковыми именами соответствующие аргументы-регистры имеют один номер значения, то эти операции вырабатывают одинаковый результат (в силу независимости семантики имени операции от контекста, см. раздел 1).

Например:

10 MOV R0 → R2
20 MOV R0 → R3

30 ADD R1 R2 → R4
40 ADD R1 R3 → R5

Очевидно, что операции 30 и 40 вырабатывают одинаковый результат и получат одинаковый номер значения. Поэтому будем поддерживать хэш-таблицу, записи в которой имеют составной ключ и номер значения. Ключ состоит из имени операции и номеров значений последовательно всех её аргументов. Итак, для операции составляем ключ. Если по этому ключу есть номер значения, то его и назначаем операции. Иначе, генерируем неиспользованный ранее номер значения, и добавляем в таблицу запись ключ-номер. Операции присваиваем этот номер.

```
EvalOper(oper)
{
   /**
    * составной ключ содержит имя операции и номера значений
    * всех аргументов операции
    */
   key = MakeCompoundKey(oper);
   /* ищем запись в хэш таблице по ключу */
   if (val_num is not found in ConstHash by key )
   {
      /* получаем ещё не использованный номер */
      val_num = GetNewValNum();
      /* и запоминаем этот номер значения */
      CreateEntryWithValNumByKey( val_num, key);
   }
}
```

При назначении номера значений операции может оказаться, что этой операции уже был назначен номер. Тогда, если уже назначенный номер и новый номер не совпадают, назначаем неопределённый номер.

```
SetOperValNum(oper, val_num)
{
   old_num = GetOperValNum(oper);
   if (old_num != UNDEF_VAL)
   {
      if (old_num != val_num )
      {
         SetOperNewValNum(oper, UNDEF_VAL);
         /* если произошли изменения, устанавливаем флаг */
         is_changed = TRUE;
      }
   }
}
```

Теперь рассмотрим, как производится обработка узла слияния. С каждым таким узлом связано множество объектов, значения которых сливаются в этом узле. Каждый объект в данном множестве имеет назначенный ему номер значения.

Функции для работы с таким множеством:

```
obj_set = GetObjsSetByJoinNode( node);

val_num = GetValNumInObjSetByObj( obj_set, obj);
```

SetValNumInObjSetByObj(obj_set, val_num);

Множество формируется следующим образом. Первоначально оно пусто. Каждый раз, когда встречается операция записи, мы обходим все узлы, входящие в её IDF и добавляем объект, в который записываем в соответствующие множества.

При обработке узла слияния обходим все объекты в указанном множестве. И для каждого объекта производим “слияние” номеров значений, приходящих от предшественников узла слияния. Получив результирующий номер значения, имитируем запись в объект, то есть вызываем SetObjValNum, тем самым запоминая факт записи результирующего номера значения в данном узле в данный объект.

```
/* === обработка узла слияния === */
EvalJoinNode( node)
{
   /* обходим множество объектов, чьи значения проходят через узел */
   foreach( obj in objs )
   {
      /* "сливаем" приходящие значения в phi_val_num */
      phi_val_num = JoinValNumsFromPredecessors( obj);
      /* имитируем запись в объект obj значения с номером phi_val_num */
      SetObjValNum( node, obj, phi_val_num);
      /* "проталкиваем" объект дальше по всем IDF узла */
      PushObjToAllIDFs( node, obj);
   }
}
```

Алгоритм внутрипроцедурного анализа:

```
do
{
   is_changed = FALSE;
   /* обход всех узлов управляющего графа */
   foreach( node in graph )
   {
      if ( not all predecessors evalued )
         continue;
      if ( GetNodeType( node) == CFG_BLOCK_JOIN )
         EvalJoinNode( node);
      else
         EvalCFGNode( node, &is_changed);
   }
   /* до тех пор, пока процесс не стабилизируется */
} while ( is_changed );
```

2.4. Доказательство корректности. Оценка сложности

Покажем, что алгоритм всегда завершает свою работу. В течение одной итерации анализа могут быть обработаны не все линейные участки: обработаны будут только те узлы, у которых уже обработаны все предшественники, не считая предшественников по обратным дугам. Когда операция обрабатывается, ей назначается какой-либо номер значения. После того, как операции назначен неопределённый номер значения, он измениться уже не может. Поэтому любая операция может находиться в одном из 3-х состоя-

ний: 1) необработанная; 2) обработанная с определённым номером значения; 3) обработанная с неопределённым номером значения. И переходы состояний возможны только из 1-го во 2-е и из 2-го в 3-е.

Алгоритм будет повторять работу до тех пор, пока не перестанут изменяться номера значений операций. Это означает, что на каждой итерации происходит переход какой-либо операции в новое состояние. Для каждой операции таких изменений не может быть больше двух. Значит, число итераций ограничено не может быть больше, чем удвоенное число операций в процедуре. Мы получили верхнюю оценку числа итераций. Но это очень грубая оценка; для более точной оценки обратимся к [4].

Алгоритм внутрипроцедурного анализа, который был приведён выше – это, по существу, модификация алгоритма (SSA-based value numbering with hashes), предложенного в [4]. В том алгоритме вводятся псевдооперации – φ-функции, имеющие аргументами значения объекта с разных ветвей управления, которые претерпевают слияние в этой φ-функции. Для алгоритма в [4] приводится и формально доказывается оценка сложности O(S×D), где S – это число операций в процедуре плюс число φ-функций. D – это средний уровень вложенности циклов; это число в любом случае не превышает S, но на практике ограничено небольшой константой.

Ключевое различие между нашим алгоритмом и алгоритмом в [4] состоит в том, что в алгоритме, представленном в [4], предполагается введение псевдоопераций φ-узлов и эти псевдооперации со своими аргументами – объектами, для которых происходит слияние значений – уже построены. В нашем алгоритме вместо псевдоопераций используются узлы слияния управляющего графа. И множество объектов, претерпевающих слияние, строится во время обхода, когда во время обработки операций записи мы обходим все IDF текущего узла. Число IDF для узла ограничено средней ветвистостью управляющего графа (число последователей или преемников у узла в среднем по всему графу). Такое число ограничено небольшой константой. К тому же обход по всем IDF достаточно провести всего один раз во время первой итерации анализа. Поэтому сложность алгоритма внутрипроцедурного анализа, представленного в данной работе, не хуже сложности алгоритма, представленного в [4]: O(S×D).

3. МЕЖПРОЦЕДУРНЫЙ ОБХОД

3.1. Общие положения

Основная идея межпроцедурного обхода заключается в том, что мы проводим анализ для всех возможных в программе ветвей исполнения, как будто срабатывают все вызовы в программе. Нам понадобится поддерживать стек вызовов. Начинаем обход с процедуры Main. Перед тем, как начать обход, помещаем процедуру в стек. Производим обход операций процедуры, как это было описано в разделе 2. Если встречается операция CALL, помещаем в стек вызываемую процедуру, и начинаем её обход. После того, как обход вызываемой процедуры завершён, извлекаем процедуру из стека и продолжаем обход исходной процедуры с операции, следующей за CALL. Когда из стека будет извлечена последняя процедура (это должна быть процедура Main), межпроцедурный обход окончен.

Использование стека позволит обнаружить рекурсивные вызовы процедур. Если не предпринимать специальных мер для рекурсивных вызовов, то по описанному выше методу межпроцедурному обхода произойдёт зацикливание. Чтобы этого избежать, для таких вызовов производится специальная обработка.

До некоторого времени будем считать, что операция CALL служит для вызова одной и только одной процедуры. Случай, когда в языке программирования, имеется возможность вызова процедуры по адресу, и заранее неизвестно какая процедура будет вызвана, будет рассмотрен в конце раздела.

3.2. Передача параметров и возврат значения

Если игнорировать факт передачи параметров из процедуры в процедуру и возможный возврат значения из процедуры, то, по существу, будет проведён внутрипроцедурный анализ для определённого множества процедур. Это означает, что пропадёт смысл межпроцедурного обхода. Поэтому следует поддерживать специальную структуру данных, в которую будут записаны номера значений параметров, которые известны к моменту вызова. При внутрипроцедурной нумерации значений в вызываемой процедуре эти номера значений будут являться инициализацией для локальных переменных, соответствующих параметрам.

В ходе внутрипроцедурного обхода может встретиться операция RET – возврат значения из процедуры. Эта операция может встретиться только в стоповом узле. Обработка такой операции будет заключаться в том, что в специально структуре будет запомнен номер возвращаемого значения. Этот номер будет являться номером значения операции CALL вызывающей процедуры.

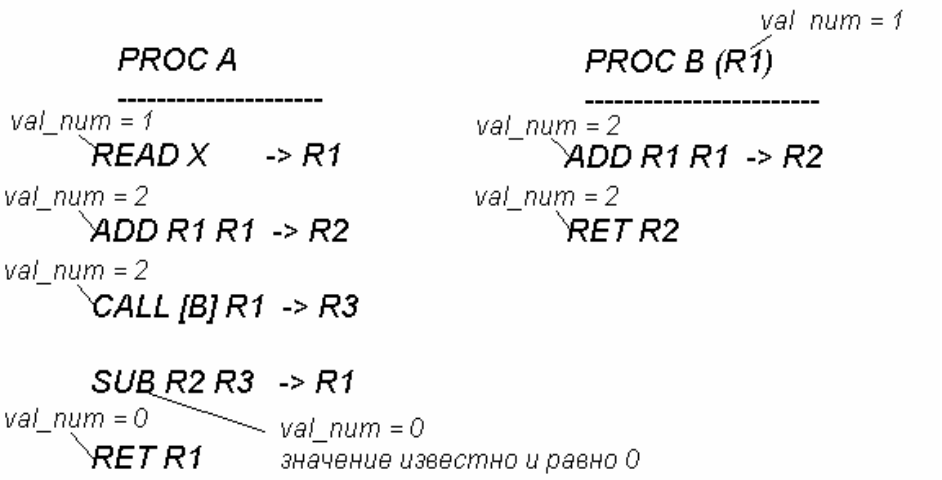

Рис. 5. Межпроцедурная нумерация значений.

3.3. Глобальные объекты. Частичная трансферная функция

Объекты делятся на локальные и глобальные (см. раздел 1). Значения глобальных объектов могут читаться и изменяться в любой из процедур, значит, они, во-первых, служат неявным средством передачи параметров в процедуру, а во-вторых, неявным средством возврата значений из процедуры.

Выше было показано, что нам понадобилась некоторая структура данных для обмена информацией между вызывающей и вызываемой процедурами. В частности, там хранились номера значений передаваемых фактических параметров.

Эту структуру будем называть *частичной трансферной функцией* (ЧТФ), и состоять она будет из:

а) списка номеров значений фактических параметров, передаваемых при вызове;

б) списка глобалов, которые используются в вызываемой процедуре и номера их значений;

в) номера значения, которое возвращает вызов (если таковое имеется);

г) списка глобалов, в которые происходит запись в вызываемой процедуре и номера их значений.

Совокупность а) и б) называется *контекстом вызова*. Совокупность в) и г) называется *эффектом вызова*. Название ЧТФ заимствовано нами из [5]. ЧТФ показывает, какой будет реакция вызываемой процедуры (какие значения в какие объекты запишутся) на заданный контекст вызова.

Помимо того, что ЧТФ – это средство передачи информации в вызываемую процедуру и обратно, это также позволит нам в определённых случаях экономить ресурсы, избегая избыточных обходов процедур. Действительно, если для некоторого контекста вызова ЧТФ уже подсчитана, то обход вызываемой процедуры необязателен, ведь эффект процедуры на заданный контекст уже известен (а значит, был когда-то вычислен, значит обход процедуры уже проводился и номера значений вызываемой процедуры уже расставлены).

3.4. Вызовы по адресу процедуры. Рекурсивные вызовы

Представим ситуацию, когда может происходить вызов процедуры по адресу. Тогда на этапе компиляции, вообще говоря, заранее неизвестно какая именно процедура вызывается. Эта проблема будет решаться заменой операции вызова по адресу на последовательность из явных операций вызова для множества процедур, которые потенциально могут в данной точке быть вызваны. В худшем случае во множество попадут все процедуры программы. Существуют методы (например, анализ указателей) позволяющие это множество сократить.

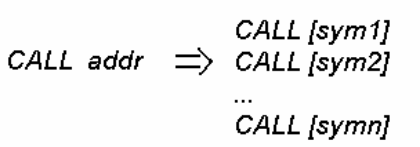

Рис. 6. Устранение вызовов по адресу.

Если вызываемая процедура уже присутствует в стеке вызовов, то нам придётся иметь дело с рекурсивным вызовом. Чтобы избежать бесконечных циклов в межпроцедурном обходе, нужно такие вызовы идентифицировать и обрабатывать особым образом. А именно, каждому объекту в эффекте ЧТФ запишем неопределённый номер значения.

```
EvalCall( oper)
{
   if ( IsRecursiveCall( oper) )
   {
      EvalRecursiveCall( oper);
      return;
   }
   context = PrepareCallContext( oper);
   if ( MatchingPTFExists( context) == FALSE )
   {
      EvalProcValNum( callee_proc);
```

```
    }
    return;
}
```

4. ПРИМЕНЕНИЕ РЕЗУЛЬТАТОВ АНАЛИЗА “МЕЖПРОЦЕДУРНАЯ НУМЕРАЦИЯ ЗНАЧЕНИЙ”

4.1. Удаление избыточных вычислений

Одно из очевидных применений результатов нумерации значений – это удаление избыточных операций.

Действительно, если две операции получили один номер значения, то это означает, что они вырабатывают одно и то же значение. Это доказано анализом уже на этапе компиляции. В таком случае будем говорить, что операции эквивалентны. Одна из операций является избыточной: её можно удалить и все использования её результата заменить на использование результата эквивалентной операции.

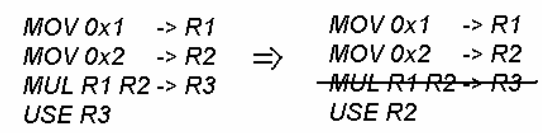

Рис. 7. Удаление избыточных вычислений.

4.2. Инлайн-подстановки процедур

Инлайн-подстановка процедур позволяет улучшить время исполнения скомпилированных программ, но при этом увеличивается размер исполняемого кода. Обычно алгоритм инлайн-подстановок старается найти компромисс, при котором достигается выигрыш производительности и увеличение размера исполняемого кода остаётся в разумных пределах. Задают некую эвристическую оценку, показывающую максимально допустимое приращение размера программы в процентах.

При заданном пороге приращения размера кода программы, чем больше подстановок будет произведено (с приоритетом для более вероятных вызовов), тем эффективнее получится код.

Под размером процедуры будем понимать число операций этой процедуры. Размер программы – это сумма размеров всех её процедур. Представим, что в какой-то момент принимается решение, подставлять ли процедуру P2 в процедуру P1. Чем размер P2 больше, тем меньше шансов состояться у данной инлайн-подстановки.

При таком подходе считаем, что после подстановки P2 в P1 sizeof(P1)=sizeof(P1) +sizeof(P2). Однако это не всегда так. Если в P1 и P2 есть эквивалентные операции, то после инлайна избыточные операции можно будет удалить. Значит, используя результаты межпроцедурной нумерации значений, можно показать, что после инлайна размер sizeof(P1)=sizeof(P1)+sizeof(P2)–num_of_common_op.

На рис. 8 слева от жирной черты показан исходный код, справа – код после применения одного шага инлайн-подстановки. Алгоритм инлайна, не учитывающий результаты анализа нумерации значений, отказывался подставлять остальные вызовы g(). Алгоритм, использующий результаты анализа, понял, что увеличения кода больше не произойдёт, и подставил все вызовы g().

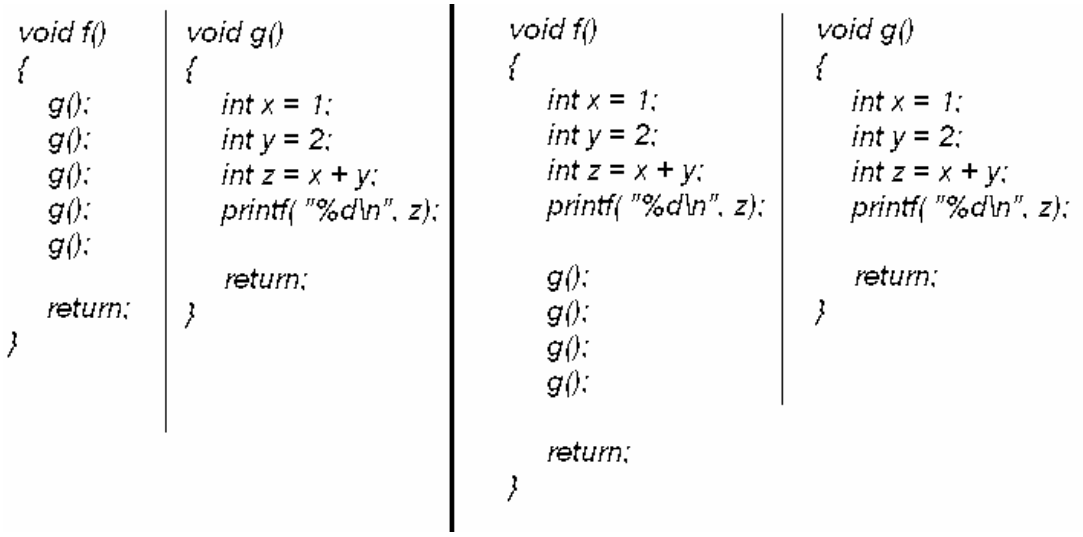

Рис. 8. Один шаг инлайн-подстановки.

Анализ “Межпроцедурная нумерация значений” позволяет более точно оценить увеличение кода процедуры, в которую производится инлайн-подстановка. Это приведёт к тому, что будет происходить больше инлайн-подстановок и будет получен более эффективный код. При этом размер программы будет оставаться в заданных пределах.

5. НЕКОТОРЫЕ ТРУДНОСТИ МЕЖПРОЦЕДУРНОГО ОБХОДА ПРОГРАММ, НАПИСАННЫХ НА ЯЗЫКЕ СИ

5.1. Нелокальные переходы

В языке Си имеется средство, позволяющее совершать нелокальные переходы из процедуры в процедуру, минуя операции CALL. Таким средством является пара вызовов setjmp и longjmp. Вызов setjmp приводит к запоминанию стека вызовов и указатель на текущую команду в структуру jmp_buf. Если затем эту структуру подать вызову longjmp, то произойдёт восстановление стека, и передача управления в точку, где находится setjmp. При этом управление может быть передано в другую процедуру.

Вызовы setjmp и longjmp нарушают идеологию межпроцедурного обхода, представленного в данной работе, поэтому приходится производить для них специальную обработку. В процессе обхода процедуры, когда встречается вызов setjmp, запоминаем в специальном стеке процедуру и узел управляющего графа.

Обработка вызовов longjmp происходит по следующему алгоритму:

```
EvalLongJump()
{
    proc = GetElemFromTheBottom( setjumps_stack);
    objs = GetObjsThatAreUsedInProc( proc);
    for ( obj = each object used in procedure proc)
    {
        foreach element in setjmp stack
        {
            Получаем номер значения объекта obj, которое было записано в объект
            к моменту, когда был вызов setjmp, и сливаем с тем номером, который
            в объекте сейчас. Записываем полученный таким образом номер значения
            в узел, в котором был вызов setjmp
```

```
    }
    return;
}
```

5.2. Обработчики сигналов

Представленный в работе межпроцедурный обход не может быть применён к программам, которые используют обработчики сигналов. Так как сигналы, как реакция на внешние события, могут поступать в произвольный момент функции обработчики могут быть вызваны асинхронно относительно выполнения программы. В теле такой функции может быть использование и запись в любые глобалы, чтение из памяти и запись в память. Программы, содержащие обработчики асинхронных сигналов, вообще говоря, не пригодны для статического анализа.

Поэтому, когда в ходе межпроцедурного обхода встречается вызов функции signal, анализ прекращается и считается, что для программы нет результатов анализа.

6. ЭКСПЕРИМЕНТАЛЬНЫЕ РЕЗУЛЬТАТЫ

Целью эксперимента было, во-первых, увидеть сами результаты анализа “межпроцедурная нумерация значений”, проводимого для реальных задач, а, во-вторых, пронаблюдать эффект, достигаемый применением этих результатов для инлайн-подстановок. Эксперимент проводился в контексте работы оптимизирующего компилятора проекта Эльбрус-3М [2, 3] на задачах пакета *Spec*95.

Табл. 1.

Результаты анализа для задач пакета Spec95

Тест	Вызывающая процедура	Вызываемая процедура	Число операций в вызываемой процедуре (N1)	Число общих операций (N2)	Процент общих операций (N2/N1*100%)
129.compress	main	compare_buffer	129	17	13
	main	fill_text_buffer	178	11	6
	main	spec_select_action	145	98	68
130.li	plist	Readone	105	11	10
	readone	Pname	165	14	8
	isnumber	Strlen	275	29	10
134.perl	main	Sprintf	1743	199	11
	Main	Myfatal	157	16	10
	Main	Instr	135	20	15

В табл. 1 показаны несколько случаев вызовов процедур и полученные для этих процедур результаты анализа. Число общих операций здесь – это число таких операций в вызывающей процедуре, которые имеют эквивалентные им операции в вызываемой процедуре. Анализ показал, что в отдельных случаях значительная часть операций в вызываемой процедуре имеет эквивалентные им операции в вызывающей процедуре. Очевидно, в таких случаях применять инлайн-подстановку выгодно.

В табл. 2 сведены показатели, демонстрирующие эффект, производимый применением результатов анализа для инлайн-подстановок. Эффект заключается как в уменьшении получаемого на выходе компилятора исполняемого кода, так и в улучшении времени ис-

полнения исполняемого кода. Размер исполняемого кода измерялся в байтах, время исполнения – в тактах архитектуры Эльбрус-3М.

Табл. 2.

Эффект от применения результатов анализа для инлайн-подстановок

Тест	Уменьшение размера исполняемого файла (size_before/size_after)	Уменьшение времени исполнения (time_before/time_after)
124.m88ksim	1,009	1,001
129.compress	1,015	1,000
130.li	1,006	1,002
134.perl	1,004	1,002
147.vortex	1,008	1,002

Результаты эксперимента показывают, что основное улучшение при использовании результатов анализа получено как уменьшение размера исполняемого кода при почти неизменившемся времени исполнения. Такой явление объясняется более эффективной работой пары оптимизаций: {инлайн-подстановка – удаление избыточных вычислений} при использовании результатов анализа.

7. ЗАКЛЮЧЕНИЕ

Анализ “Межпроцедурная нумерация значений” является обобщением известного механизма “Нумерация значений” и выведением последнего на качественно новый межпроцедурный уровень.

В статье рассмотрена модификация алгоритма, предложенного в [4], для проведения анализа нумерация значений внутри одной процедуры. Затем предложены основные принципы для проведения межпроцедурного анализа нумерация значений, опираясь на внутрипроцедурный анализ. Показаны некоторые проблемы, возникающие при межпроцедурном обходе, и пути их решений. Указаны области применения результатов анализа для проведения оптимизаций.

В настоящее время механизм межпроцедурного анализа и применение результатов анализа для инлайн-подстановок процедур реализованы авторами статьи полностью в языковом оптимизирующем компиляторе ecf_opt, компании Эльбрус [2, 3].

ЛИТЕРАТУРА

1. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ulman. Compilers: Principles, Techniques and Tools. Addison-Wesley Reading, 1986.

2. ЗАО МЦСТ. Официальный сайт http://www.mcst.ru.

3. Diefendorf K. The Russians Are Coming: Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee // Microprocessor Report, vol 11, № 2, February 15, 1999. P. 1-7.

4. Simpson, Loren Taylor. Value-Driven Redundancy Elimination. Ph.D. Thesis, Rice University, Houston, Texas, 1996.

5. Wilson, Robert Paul. Efficient, Context-Sensitive Pointer Analysis For C Programs. Ph.D. Thesis, Stanford University, 1997.