Эффективный алгоритм преобразования потока управления в поток данных

Дроздов А. Ю., Новиков С. В., Шилов В.В.
Институт микропроцессорных вычислительных систем РАН
sasha@mcst.ru, novikov@mcst.ru

Введение

В работе описан метод преобразования программ в предикатную форму, основанный на *gated single assignment* (GSA) форме представления [1] потока данных программы. Предикатное представление программы широко используется оптимизирующими компиляторами архитектурных платформ с явно выраженной параллельностью на уровне отдельных операций и поддержкой предикатного и спекулятивного режимов исполнения операций. Для обозначения такого рода архитектур используются термины *EPIC (Explicitly Parallel Instruction Computing)* или *архитектура с явно выраженной параллельностью на уровне команд* [2-6].

Одной из основных задач, которую должен решать оптимизирующий компилятор для платформ с поддержкой предикатных вычислений, является построения предиката операции. Предикатными вычислениями называется условное исполнение команд, зависящих от значения условного операнда данной команды, называемого предикатом. Когда значение предиката есть истина (TRUE), команда исполняется нормально; а когда значение предиката есть ложь (FALSE), 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. При переносе операции выше нескольких переходов, предикат операции должен быть вычислен на основании условий этих переходов.

Приведем пример, в котором благодаря предикатным вычислениям был удален переход. На рис. 1 приведено преобразование безусловных вычислений в предикатный код. Условие для операции будем записывать в квадратных скобках рядом с операцией. Для каждого условия существует возможность задания маски, с которой оно влияет на операцию. Если условие используется явно, то маска равна TRUE и записывается как .t. Если условие используется инвертировано, то маска равна FALSE и записывается как .f.

Исходный тест	Безусловный код	Предикатный код

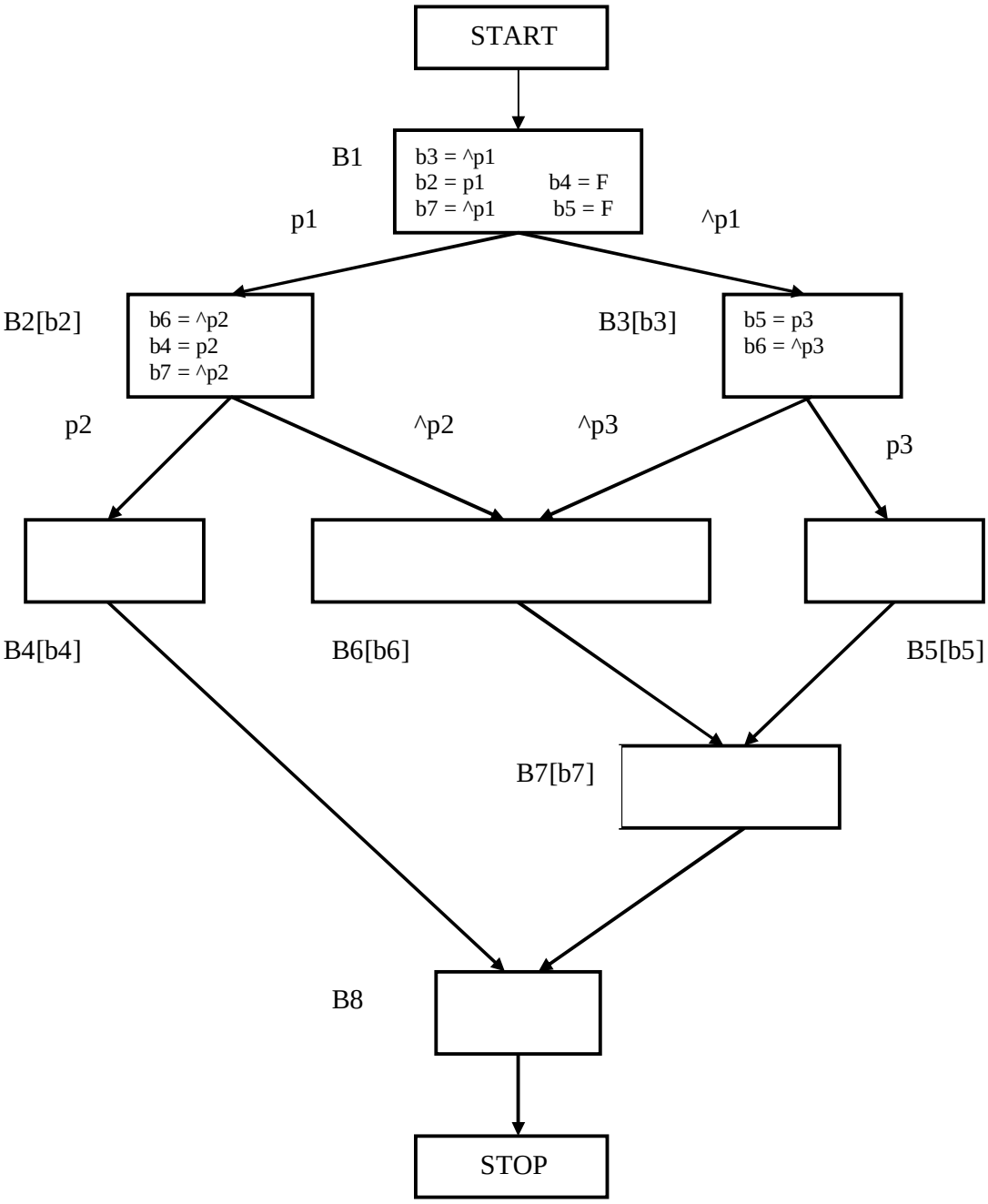
if (a = = b)
{
  a = a + 1;
} else
{
  b = b + 1;
}
	cmp a, b → P
	branch (1) [P.t]
	a = a + 1
	1 b = b + 1
		cmp a, b → P
		a = a + 1 [P.t]
		b = b + 1 [P.f]

Рис. 1. Предикатные вычисления.

Процесс перевода безусловного кода в предикатный традиционно называется if-conversion ([7, 8]). По сути, if-conversion преобразует зависимости по управлению с операциями переходов в зависимости по данным с предикатами. Данное преобразование выполняется на ациклических участках программ. Один из подходов выполнения преобразования if-conversion описан в работе [9]. При данном подходе каждому классу CD-эквивалентных узлов назначается свой предикат. В узлах, соответствующих CD-дугам этих классов строятся инициализации предикатов (рис. 2). Необходимые начальные

инициализации достраиваются перед операциями записи в предикаты. Например, для узла B6 рис. 2 CD-дугами являются дуги (B2, B6) и (B3, B6). Поэтому в узлах B2 и B3 находятся инициализации условия b6. При преобразовании if-conversion операции инициализации предикатов должны быть поставлены под условия участков, на которых они создаются. Для некоторых условий может сложиться ситуация, когда они достигнут своего использования в неинициализированном виде. В этом случае перед всеми записями в предикаты достраиваются начальные инициализации предикатов значением FALSE. Например, для узла B4 условие инициализируется под предикатом b2 (b4=p2[b2]) и в случае, когда b2=FALSE, предикат b4 окажется неинициализированным. Поэтому в узле B1 есть начальная инициализация предиката b4 значением FALSE. Недостаток этого подхода состоит в появлении зависимостей по предикатам для условных операций, что не позволяет достигать хорошей параллельности на уровне операций. Кроме этого, данный подход требует большого количества дополнительных операций на вычисление предикатов.

В работе предлагается метод построения предикатов для ациклических участков управляющего графа, обеспечивающий возможность распараллеливания условных операций, а также требующий меньшее количество дополнительных операций при вычислении предикатов. В основе метода лежит предикатная форма единственного присваивания представления потока данных программы (Gated Single Assignment, GSA). В работе описывается GSA-форма и приводится алгоритм ее построения. Показывается, как на основе GSA-формы можно построить предикатное выражение, которое описывает предикатный путь любого узла ациклического участка программы относительно произвольного предшественника этого узла из того же ациклического участка. В работе описываются способ реализации предикатных выражений, который был использован в оптимизирующем компиляторе проекта “Эльбрус” [11], а также способ хеширования предикатных выражений, позволяющие избегать дублирования при их построении.



p1 = …; b5 = FALSE; b4 = FALSE
b2 = p1; b3 = ^p1; b7 = ^p1
p2 = …[b2]; p3 = …[b3]
b6 = ^p2[b2]; b4 = p2[b2]; b5 = p3[b3]; b6 = ^p3[b3]; b7 = ^p2[p2]

Рис.2. Построение предикатов методом инициализаций в узлах с выходящими CD-дугами.

1.Описание пути в программе

GSA-форма является расширением Static Single Assignment (SSA) формы представления потока данных программы [10]. В SSA-форме в точках схождения управляющего графа строятся однотипные φ-функции для связи различных определений переменной, достижимых от входных дуг точек схождения. При этом отсутствует информация о том, при каких условиях данные определения достигают точек схождения управления. В GSA-форме такая информация появляется. Для отображения условий достижимости определениями точек схождения в GSA-форме вводятся несколько типов условных функций различных классов точек схождения:

1. γ-функция – функция точки схождения участка if-then-else. Функция имеет вид X3=γ(B, X1, X2), где B – условие перехода конструкции if-then-else, X1 – определение, соответствующее условию B, а X2 – определение, соответствующее инверсному условию ^B.
2. μ-функция – функция, соответствующая голове цикла. Функция имеет вид X2=μ(X0, X3), где X0 – определение, соответствующее входу в цикл, X3 – определение в цикле.
3. η-функция определяет значение переменной на выходе цикла.

Для целей данной работы будет интересна только γ-функция, отображающая условие достижения точек схождения в ациклических участках.

Для описания пути в управляющем графе используется свойство доминирования, которое на момент построения γ-функций считается вычисленным и отображенным в управляющем графе в виде дерева доминаторов. По определению узел w до*минирует* узел v (обозначеие dom(v)), если любой путь от стартового узла управляющего графа до узла v проходит через узел w. Если w не совпадает с узлом v, то узел w *строго доминирует* узел v. Отношение доминирования транзитивно и может быть представлено в виде дерева на узлах управляющего графа с вершиной в стартовом узле[12].

Назовем *предикатным путем* узла N путь в CFG графе от непосредственного предшественника в дереве доминаторов idom(N) до узла N, содержащий только преемники узла idom(N). Иными словами, все узлы на пути от idom(N) до N доминируются узлом idom(N).

В управляющем графе CFG (N, E) мы можем рассматривать любой путь как набор дуг из множества E. Введем обозначения для возможности описывать предикатный путь узлов.

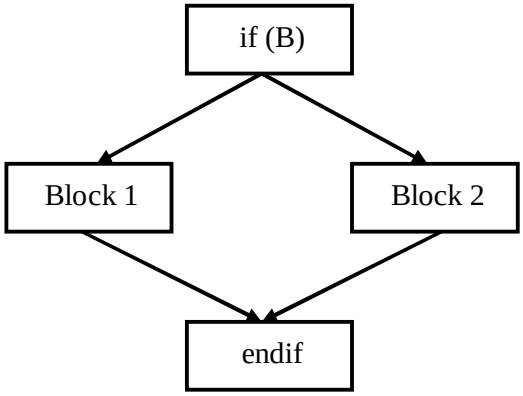


Рис. 3. Условное ветвление.

Для безусловных дуг будем использовать обозначение Λ. Например, так будут обозначены дуги от узлов Block 1 и Block 2 к узлу endif. Тогда P(Block 1,endif)=Λ, P(Block 2, endif)=Λ. Для условных дуг обозначения будем делать с указанием условия, соответствующего этим дугам. Для ветвления if (B) обозначим выходящие дуги как B.t и B.f. В нотации γ-функций условные дуги будут выглядеть следующим образом: B.t=γ(B, Λ, Ø) и B.f=γ(B, Ø, Λ). Здесь символ Ø обозначает дугу, которую не рассматривают в данный момент. Для упрощения описаний предикатных путей, представленных в описанной выше нотации используются следующие правила:

R = R1∪ R2 : **case** R1 = = Ø
return R2
case R2 = = Ø
return R1
case R1 = = γ(B, R1.t, R1.f) **and** R2 = = γ(B, R2.t, R2.f)
return γ(B, (R1.t ∪ R2.t), (R1.f ∪R2.f))

R = R1•R2 **case** (R1 = = Ø) **or** (R2 = = Ø)
return Ø
case R1 = = Λ
return R2
case R2 = = Λ
return R1
case R1 = = γ(B, R1.t, R1.f)
return γ(B, (R1.t • R2), (R1.f • R2))

Применяя правила к описателям дуг в примере, получаем:

p.t(if, endif) = γ(B, Λ, Ø)•Λ = γ(B, Λ, Ø)
p.f(if, endif) = γ(B, Ø, Λ)•Λ = γ(B, Ø, Λ)
P(if, endif) = p.t(if, endif) ∪ p.f(if, endif) = γ(B, Λ, Ø) ∪ γ(B, Ø, Λ) = γ(B, Λ, Λ)

Предикатным выражением будем называть описатель пути в ациклическом участке.

Приведем алгоритм построения γ-функций, который получается путем усечения более общего алгоритма работы [9]. Алгоритм работает на ациклическом участке управляющего графа, на котором построено дерево доминаторов. На участке возможны только условные конструкции типа if-then-else. На управляющем графе заданы прямая и обратная нумерации [12]. В западной литературе эти нумерации называются depth-first numbering (DFN) [10] и reverse post order numbering (RPO) [10].

Алгоритм 1

GP(n) – описатель предикатного пути для узла n;
ancestor(n) – ссылка на предшественника в дереве доминаторов;
p(u, v) – предикатное выражение, соответствующее пути (u,v) в дереве доминаторов, если путь (u, v)=u→u1→…→uN→v, то p(u,v)=GP(u)•GP(u1)•…•GP(uN)•GP(v);
P_ancestor(n) – предикатное выражение p(ancestor(n),n).

```
main ()
{
 1:  for each node u from acyclic region in reverse dfn
 2:    for each dominator tree children v of u
 3:      for each predecessor edge e = (w,v)
 4:        if w = = u
 5:          GP(v) ← GP(v) ∪ (e)
 6:        else
 7:          p(subroot(w),v) ← Eval( e)
 8:          add p(subroot(w),v) to ListP(v)
 9:        endif
 10:     endfor
 11:   endfor
 12:   for each dominator tree children v of u in rpo
 13:     for each p(subroot(w),v) from ListP(v)
 14:       GP(v) ← GP(v) ∪ p(subroot(w),v)
 15:     endfor
 16:     save GP(v) as gated path of v
 17:     ancestor(v) ← u
 18:   endfor
 19: endfor
} /* main */

/**
 * Процедура построения выражения пути, соответствующего дуге e = (w,v)
 * subroot(w) = ancestor(w1);  w1 = ancestor(w2);  ...  wK = ancestor( w)
 */
Eval ( Edge e)
{
1:   n ← GetPred( e)
2:   while n != 0
3:     add n to begin of worklist
4:     n ← ancestor(n)
5:     root ← n
6:   endwh
7:   for each n from begin of worklist
8:     if cur_node = = 0
9:       cur_node ← n
10:    else
11:      P_res ← GP(cur_node) • P_ancestor(n)
12:      /* Компрессия пути дерева доминаторов */
13:      save P_res as P_ancestor(n)
```

```
14:         ancestor(n) ← root
15:       endif
16:    endfor
17:    P_res ← P_res • (e)
18:    return (P_res)
} /* eval */
```

2. Преобразование γ-функции в предикатное выражение

После построения каждая γ-функция имеет структуру графа. Узлом графа является элементарная γ-функция вида G(**P**, **arg1**, **arg2**), где **P** – предикат условного перехода программы, а **arg1** и **arg2** – аргументы γ-функции, тоже являющиеся γ-функциями. Аргументы можно трактовать как преемники узла в графе γ-функции. Существует два типа терминальных γ-функций – Λ и Ø, у которых нет аргументов. Для задания правил преобразования в предикатное выражение γ-функций, представленных в виде графа, введем понятие эквивалентности γ-функций. Две нетерминальные γ-функции эквивалентны, если предикаты γ-функций совпадают, а аргументы эквивалентны. Алгоритм анализа эквивалентности тривиален: все γ-функции получают номера, и если номера совпадают, то соответствующие γ-функции эквивалентны.

Алгоритм 2

```
main ( )
{
1:    marker ←  GetGammaNewMarker( )
2:    set  Ø class 0
3:    set Λ class 1
4:    CurClassNum ← 2
5:    mark Ø by marker
6:    mark  Λ by marker
7:    for each  γ-function G
8:       SetGammaClassRecur( G)
9:    endfor
} /* main */

/**
 * Процедура установки классов эквивалентности для γ-функций.
 */
SetGammaClassRecur( γ-function G)
{
1:    if  G is marked by marker
2:         return
3:    endif
4:    mark G by marker
5:    arg1 ← GetGammaArg1( G)
6:    arg2 ← GetGammaArg2( G)
7:    SetGammaClassRecur( arg1)
8:    SetGammaClassRecur( arg2)
9:    class_num1 ← GetGammaClassNum( arg1)
10:   class_num2 ← GetGammaClassNum( arg2)
11:   P ← GetGammaPredct( G)
12:   entry ← hashing  G by P, class_num1, class_num2
13:   if entry is new
14:      SetGammaClassNum( G, CurClassNum)
15:      CurClassNum++
16:   else
```

```
17:       SetGammaClassNum( G, GetEntryClassNum( entry))
18:    endif
19:    return
} /* SetGammaClassRecur */
```

После того, как были определены эквивалентные γ-функции можно запускать алгоритм построения предикатных выражений по графу γ-функции. Для выполнения такого преобразования используются следующие правила:

1. $G(q, \Lambda, \Lambda) \rightarrow \Lambda$
2. $G(q, G(p, 1, 2), 2) \rightarrow G(q\&p, 1, 2)$
3. $G(q, 1, G(p, 1, 2)) \rightarrow G(\hat{}(\hat{}q\&\hat{}p), 1, 2)$
4. $G(q, G(p, 1, 2), 1) \rightarrow G(\hat{}(p\&\hat{}q), 1, 2)$
5. $G(q, 1, G(p, 2, 1)) \rightarrow G(\hat{}(q\&\hat{}p), 1, 2)$
6. $G(q, G(p, 1, 2), G(r, 1, 2)) \rightarrow G(\hat{}(\hat{}(q\&p)\&\hat{}(\hat{}(q)\&r)), 1, 2)$
7. $G(q, G(p, 1, 2), G(r, 2, 1)) \rightarrow G(\hat{}(\hat{}(q\&p)\&\hat{}(\hat{}(q)\&\hat{}(r))), 1, 2)$

В правилах используются обозначения:
1, 2 – классы эквивалентности аргументов γ-функции;
q – предикат γ-функции;
p – предикат γ-функции;
& – операция И над предикатами:
0 & 0 = = 0
1 & 0 = = 0
0 & 1 = = 0
1 & 1 = = 1
^ – инверсия предикатов.

На основании приведенных правил работает алгоритм преобразования:

Алгоритм 3.

```
main ( )
{
1:   change ← TRUE
2:   while  change = = TRUE
3:      marker ← GetGammaNewMarker( );
4:      Change ← FALSE;
5:      ReduceGammaRecur( G);
6:   endwh
} /* main */

ReduceGammaRecur( γ-function G)
{
1:   if  G is marked by marker
2:      return
3:   endif
4:   mark G by marker;
5:   if  is G pattern N
6:     apply G pattern N
7:     Change ← TRUE
8:   endif
```

```
9:    if Change = = TRUE
10:       return
11: endif
12: ReduceGammaRecur( GetGammaArg1( G));
13: if Change = = TRUE
14:      return
15: endif
16: ReduceGammaRecur( GetGammaArg1( G));
17: return
} /* ReduceGammaRecur */
```

Правила, на которых основан алгоритм преобразования, являются полным набором вариантов конструкции if-then-else в нотации γ-функций. При каждом обходе графа γ-функции будет срабатывать одно из правил, пока γ-функция не будет иметь вид G (path_expr, TERM1, TERM2), где path_expr – предикатное выражение, состоящее из предикатных операций И и инверсии предикатов, а TERM1 и TERM2 – терминальные γ-функции, причем TERM1 и TERM2 не совпадают. Для γ-функции справедливо тождество G(path_expr, TERM1, TERM2) = = G(^path_expr, TERM2, TERM1).

3. Построение предикатного выражения

Предикатные выражения удобно сохранять в виде списочной структуры (рис. 4).

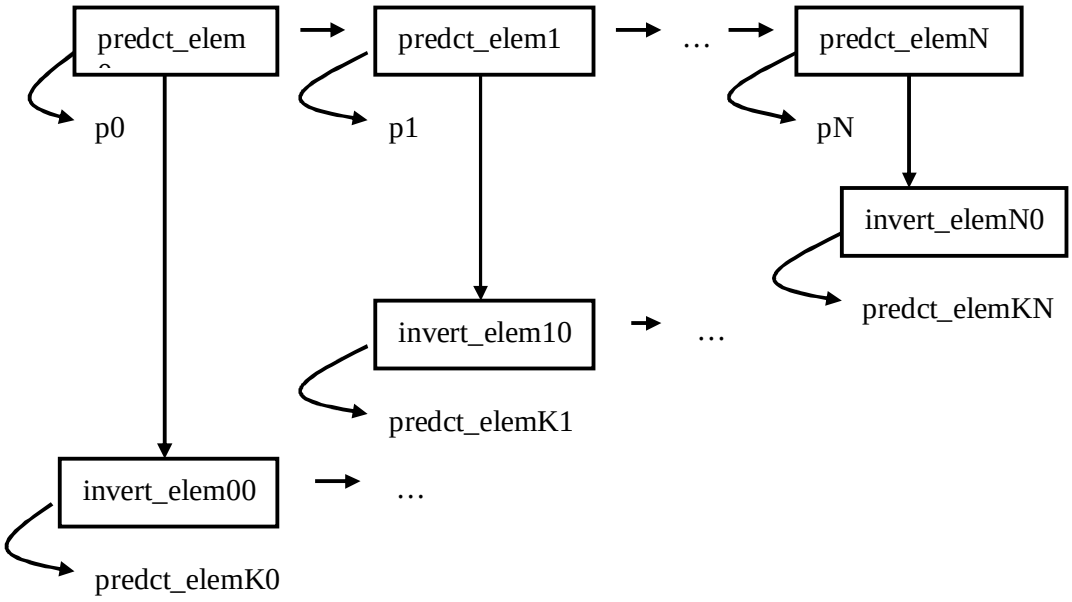


Рис. 4. Представление предикатного выражения в виде списочной структуры.

В элементе списка predct_elem содержится информация о предикате и о том, как установлены скобки инверсий, открывающиеся с данного элемента списка. В элементе инверсий содержится информация о закрывающей скобке инверсии. Например, выражение p1&^(p2&^(p3)) будет представлено в виде структуры рис. 5.

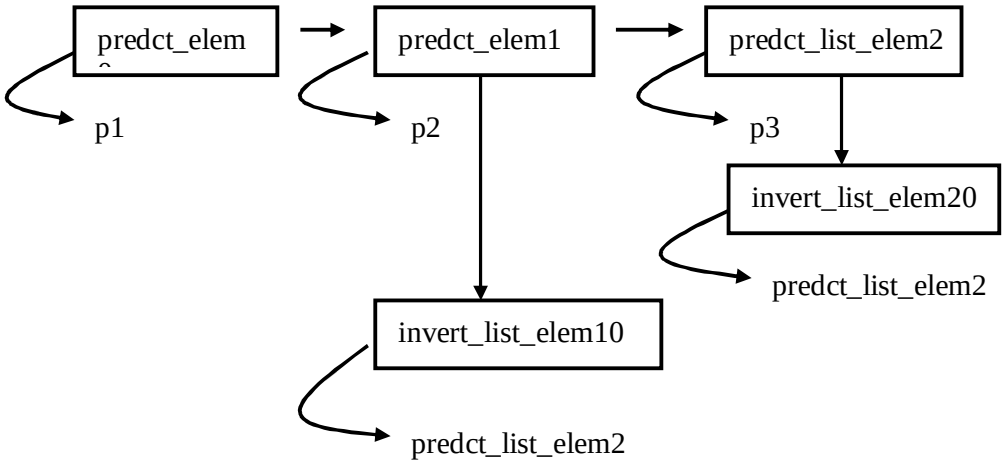


Рис. 5. Реализация предикатного выражения p1&^(p2&^(p3)) в виде списочной структуры.

В архитектуре с поддержкой предикатных и спекулятивных вычислений важно уметь эффективно строить предикатное выражение для каждого узла управляющего графа, что позволяет переводить вычисления в предикатную форму и избавляться от избыточных переходов. Для реализации предикатного выражения в архитектуре "Эльбрус" [11] используется операция предикатной конкатенации с подавлением дефектности: **land** p1 p2 → p3. Для аргументов операции **land** должны быть заданы маски предикатов, с которыми они используются в операции. Например, операция **land** p1.f p2.f → p3 сначала инвертирует аргументы p1 и p2, а затем выполняет их конкатенацию. Семантика операции такова, что если первый аргумент с учетом его маски равен FALSE, то значение второго операнда игнорируется и не приводит к прерыванию работы программы. Таким образом, если порядок предикатов в предикатном выражении соответствует порядку их следования в исходной программе, то с использованием операции **land** материализация предикатного выражения становится тривиальной. Алгоритм преобразования γ-функции в предикатное выражение сохраняет порядок вычисления предикатов. Приведем алгоритм построения операций **land**, соответствующих описателю предикатного выражения, представленного в виде списочной структуры.

Алгоритм 4.

```
ConstrGammaPredctExpr( γ-function G)
{
1:   OpenBracketsTbl ← NewTbl( )
2:   CloseBracketsTbl ← NewTbl( )
3:   Stack ← NewStack( )
4:   for each predct_list_elem in  G
5:      if invert list of predct_list_elem is not empty
6:         /**
7:          * Занесение элементов списка предикатного выражения
8:          * в таблицу открывающих скобок и таблицу закрывающих скобок.
9:          * Процедура AddPredctListElemToTbl увеличивает счетчик занесений
10:         * элемента списка в таблицу.
11:         */
 12:        for each invert_list_elem of predct_list_elem
 13:           AddPredctListElemToTbl( predct_list_elem, OpenBracketsTbl)
```

```
14:           close_bracket ← GetInvertListElemCloseBracket( invert_list_elem)
15:           AddPredctListElemToTbl( close_bracket, CloseBracketsTbl)
16:         endfor
17:       endif
18:       /**
19:        * Занесение элемента списка в вершину стека.
20:        * По элементу списка устанавливается в вершине стека адрес предикатного
21:        * аргумента и его маска. Маска изначально устанавливается в TRUE.
22:        */
23:       PushPredctListElemToStack( Stack, predct_list_elem)
24:       if predct_list_elem is not in CloseBracketsTbl
25:           continue
26:       endif
27:       /**
28:        * Цикл для всех закрывающих скобок, соответствующих predct_list_elem.
29:        */
30:       while predct_list_elem in CloseBracketsTbl
31:           prev_arg ← 0
32:           while ( 1 )
33:             /**
34:              * Инициализация аргумента по информации из вершины стека.
35:              */
36:             cur_arg ← GetArgByStackTopElem( Stack)
37:             if  prev_arg != 0
38:               /**
39:                * Построение операции land по аргументам cur_arg и prev_arg.
40:                * Процедура построения инициализирует аргумент для дальнейшей
41:                * материализации предикатного выражения.
42:                */
43:               prev_arg ← ConstrOperLAND( cur_arg, prev_arg)
44:             else
45:               prev_arg ← cur_arg;
46:             endif
47:             /**
48:              * Получение элемента списка предикатного выражения из вершины стека.
49:              */
50:             cur_predct_list_elem ← GetPredctListElemByStackTopElem( Stack)
51:             if  cur_predct_list_elem is in OpenBracketsTbl
52:                 /**
53:                  * Устанавливаем в вершине стека новый адрес и новую маску предиката
54:                  * по предикатному аргументу.
55:                  */
56:                 SetArgToStackTopElem( Stack, prev_arg)
57:                 break
58:             endif
59:             /**
60:              * Удаление элемента из вершины стека.
61:              */
62:             PopStackTopElem( Stack)
63:           endwh
64:
65:           /**
66:            * Инверсия предиката, находящегося в вершине стека. Инвертируется значение
67:            * маски предикатного аргумента, сохраняемого в элементе стека.
68:            */
69:           InvertArgInStackTopElem( Stack)
70:
71:           /**
72:            * Удаление открывающей скобки осуществляется удалением cur_predct_list_elem
73:            * из таблицы OpenBracketsTbl.
74:            * Удаление закрывающей скобки осуществляется удалением predct_list_elem
75:            * из таблицы CloseBracketsTbl.
```

```
76:          * При удалении уменьшается счетчик вхождений элемента
77:          * в таблицу. Когда счетчик становится равным 0, элемент удаляется
78:          * из таблицы.
79:          */
80:         DeletePredctListElemFromTbl( cur_predct_list_elem; OpenBracketsTbl)
81:         DeletePredctListElemFromTbl( predct_list_elem, CloseBracketsTbl)
82:      endwh
83: endfor
84: /**
85:  * Оставшиеся элементы в стеке не требуют инверсий.
86:  * Осуществляем их конкатенацию.
87:  */
88: prev_arg ← 0
89: while  Stack is not empty
90:      cur_arg ← GetArgByStackTopElem( Stack)
91:      if prev_arg != 0
92:          prev_arg ← ConstrOperLAND( cur_arg, prev_arg)
93:      else
94:          prev_arg ←cur_arg
95:      endif
96:      PopStackTopElem( Stack)
97: endwh
98: return
} /* ConstrGammaPredctExpr */
```

Реализация предикатного выражения γ-функции позволяет построить полный предикат узла относительно любого его предшественника в дереве доминаторов. Пусть узел D – предшественник узла N в дереве доминаторов, тогда между узлами D и N существует путь D→D1→D2→...→ N. Предикатное выражение для узла N может быть построено как конкатенация предикатных выражений γ-функций узлов пути дерева доминаторов: predct_expr(D, N)=GP(D1)&GP(D2)&...&GP(N).

4. Хеширование предикатного выражения

Для того, чтобы избежать дублирования операций при реализации предикатных выражений для разных узлов управляющего графа, можно применить хеширование. Ключом для хеширования предикатного выражения может быть строка слов, элементом которой может являться либо номер предиката, либо открывающая скобка инверсии, либо закрывающая скобка инверсии. Например, для выражения ^(^(1)&^(2)), где 1 и 2 – номера предикатов, ключом будет строка OPEN-OPEN-1-CLOSE-OPEN-2-CLOSE-CLOSE, в которой, например, OPEN= -1, а CLOSE= -2.

Приведем алгоритм формирования ключа для хеширования по списочной структуре предикатного выражения.

Алгоритм 5.

```
CreatePredctExprHashVect( γ-function G)
{
1:   for each predct_list_elem in G
2:      for each invert_list_elem of predct_list_elem
3:         /**
4:          * Пакуем в вектор число, соответствующее открывающей скобке.
5:          */
6:         PackInt( res_vect, OPEN)
7:         close_bracket ← GetInvertListElemCloseBracket( invert_list_elem)
8:         AddPredctListElemToTbl( close_bracket, CloseBracketsTbl)
9:      endfor
10:     /**
```

```
11:       * Пакуем число, соответствующее предикату элемента списка.
12:       */
13:      PackInt( res_vect, GetPredctListElemPredctKey( predct_list_elem))
14:      while predct_list_elem in CloseBracketsTbl
15:         /**
16:          * Пакуем в вектор число, соответствующее закрывающей скобке.
17:          */
18:         PackInt( res_vect, CLOSE)
19:         DeletePredctListElemFromTbl( predct_list_elem, CloseBracketsTbl)
20:      endwh
21:   endfor
22:   return
} /* CreatePredctExprHashVect */
```

5. Предикатное выражения произвольного пути

Предикатное выражение, соответствующее γ-функции, определяет предикат узла относительно его непосредственного доминатора. На основе предикатного выражения γ-функции можно также построить предикатное выражения для узла относительно любого предшественника по управлению этого узла, лежащего на пути между непосредственным доминатором и узлом. Для преобразования исходного предикатного выражения в выражение для произвольного пути необходима информация о достижимости вниз для узлов региона, где вычисляются выражения. При ее наличии предикатное выражение для узла относительно произвольного узла, лежащего на пути между непосредственным доминатором и самим узлом, получается путем исключения из предикатного выражения γ-функции тех предикатов, узлы которых недостижимы вниз от выбранного произвольного узла. Например, предикатное выражение узла Node 4 (рис. 6) относительно узла Node 1, построенное алгоритмом, имеет вид: ^(p1&&p2)&&^(^(p1)&&p3). Предикатное выражение узла Node 4 относительно узла Node 2 получаем исключением предикатов, узлы которых недостижимы вниз от узла Node 2 (предикаты p1 и p3): ^(p2). Предикатное выражение узла Node относительно узла Node 3 получаем исключением предикатов, узлы которых недостижимы вниз от узла Node 3 (предикаты p1 и p2): ^(p3). Таким образом, если переносить операцию из узла Node 4 в узел Node 2, то операцию нужно поставить под условие ^(p2), а если в узел Node 3, то под условие ^(p3). Корректность этих условий очевидна из рис. 6.

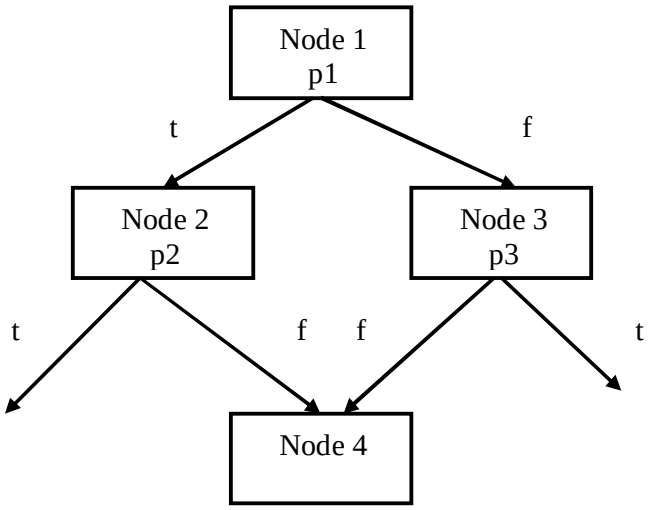


Рис. 6. Фрагмент управляющего графа

6. Пример работы алгоритма

На рис. 7 показаны результаты работы алгоритмов, строящих предикаты для узлов управляющего графа (рассмотрен ациклический участок графа, содержащий только условные конструкции типа if-then-else). Построенные исходным алгоритмом операции могут быть сгруппированы в 4 последовательные группы, внутри каждой из которых операции не зависят друг от друга. Итоговое количество операций равно 13, из них 3 исходных и 10 дополнительных. Результаты работы предлагаемого метода оказываются предпочтительнее: последовательных групп получается только 3, а количество дополнительных операций равно 6. Количество последовательных групп определяет степень параллельности вычислений на уровне операций, поэтому можно сказать, что в данном примере предложенный метод на 25% улучшает параллельность вычислений. Также почти в 2 раза уменьшены накладные расходы на вычисления условий для узлов ациклического участка.

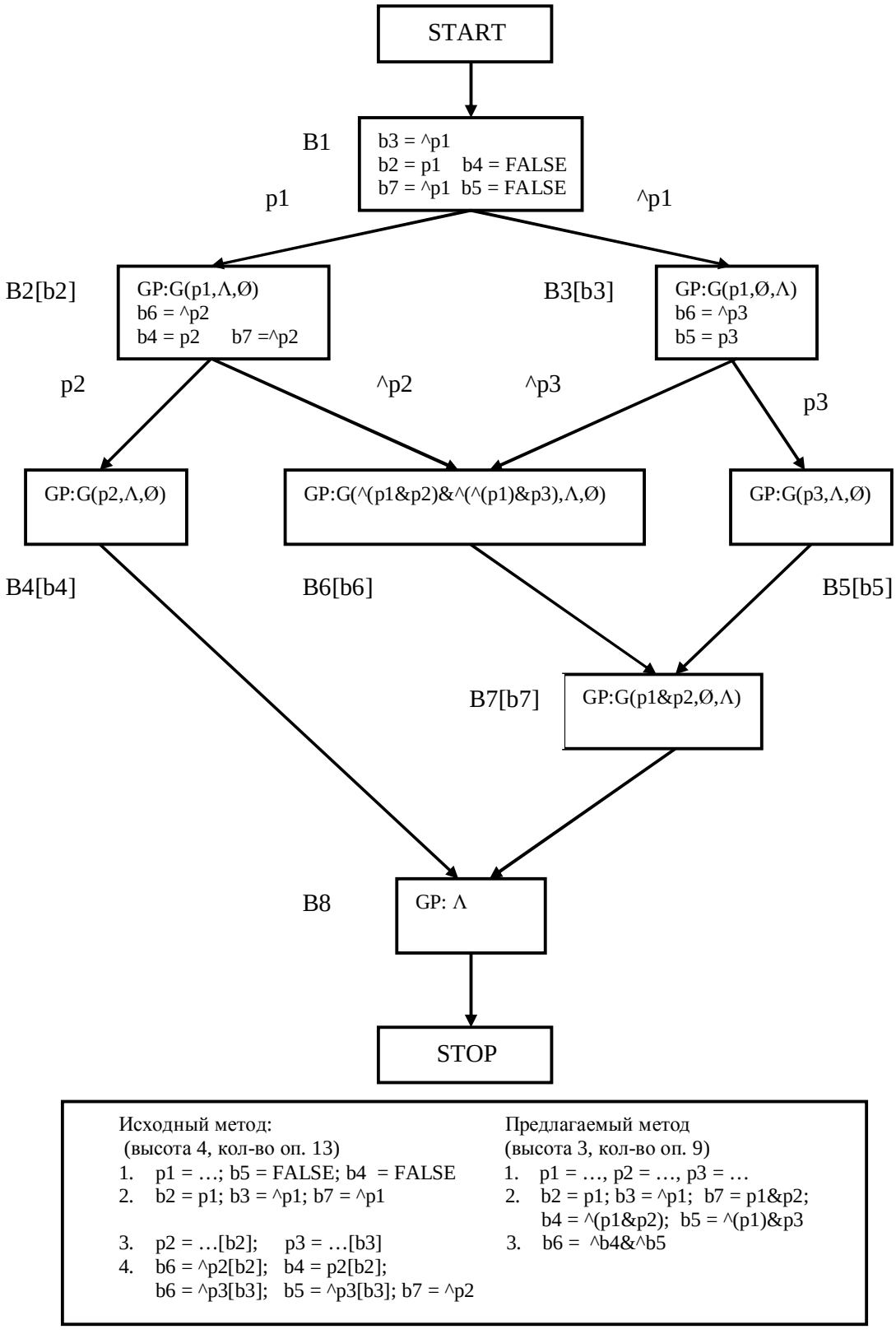


Рис.7. Построение предикатов двумя методами.

Заключение

В работе был описан эффективный метод построения предикатов. Данный метод используется оптимизирующим компилятором проекта “Эльбрус” для выполнения оптимизирующих и технологических преобразований, требующих построения условий для операций. Работоспособность метода была проверена на всей тестовой базе оптимизирующего компилятора. Предложенный подход позволил наиболее полно использовать возможности EPIC-архитектуры E2k за счет обеспечения хорошей параллельность на уровне операций и за счет небольших накладных расходов на предикатные вычисления при преобразованиях.

Список литературы

1. **Ballance R., Maccabe A.,** and **Ottenstein K.** The Programm Dependence Web: a Representation Supporting Control- Data- and Demand-Driven Interpretation of Imperative Languages // In Proceedings of the SIGPLAN’90 Conference on Programming of Languages Design and Implementation, June 1990. P. 257-271.
2. **August D. I., Crozier K. M., Sias J. W., Eaton P. R., Olaniran Q. B., Connors D. A.,** and **Hwu W. W.** The IMPACT EPIC 1.0 Architecture and Instruction Set reference manual: Technical Report IMPACT-98-04 / IMPACT, University of Illinois, Urbana, IL, February 1998.
3. **M. S. Schlansker, B. R. Rau.** EPIC: An Architecture for Instruction-Level Parallel Processors: Technical Report HPL-1999-111 – Compiler and Architecture Research Hewlett-Packard Laboratories, Palo Alto, February 2000.
4. **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.
5. **Intel Itanium 2 Processor** Reference Manual, Document Number: 251110-001, June 2002.
6. **NArch** Architecture Specification. Draft D 1.2.1 – Moscow Center of SPARC-technology, 1996.
7. **David I. August, Wen-mei W. Hwu,** and **Scott A. Mahlke.** A Framework for Balancing Control Flow and Predication // Proceedings of the 30th annual IEEE/ACM International Symposium on Microarchitecture. December, 1997. P. 92-103.
8. **Joseph C. H. Park; Mike Schlansker**. On Predicated Execution – Software and System Laboratory HPL-91-58, May, 1991.
9. **Pend Tu, David Padua.** Efficient Building and Placing of Gating Functions – Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaingn, 1995.
10. **Steven S. Muchnick.** Advanced Compiler Design and Implementation – Morgan Kauffman, San Francisco, 1997, chapter 7.2.
11. **Babayan B. A**. E2k Technology and Implementation. // Proceedings of the Euro-Par 2000 – Parallel Processing: 6th International. – V. 1900/2000. – January, 2000. – P. 18-21.
12. **Касьянов В. Н.; Евстигнеев В. А.** Графы в программировании: обработка, визуализация и применение. – СПб.: БХВ-Петербург, 2003, главы 2, 10.