Анализ предикатных выражений и его использование в оптимизирующих компиляторах для архитектур с явно выраженным параллелизмом

Дроздов А. Ю., Кирнасов А.Е.

ИМВС РАН, г. Москва

sasha@mcst.ru

Введение

Методы анализа предикатов имеют долгую историю развития и применения в оптимизирующих компиляторах. Изначально эти методы в основном использовались для уточнения потокового анализа программ. Это достигалось путем включения в анализ потоков данных и управления механизмов, позволяющих отслеживать влияние на них предикатных выражений программы [1]. С появлением архитектур процессоров типа VLIW (Very Long Instruction Word) и *EPIC (Explicitly Parallel Instruction Computing)* [2,3,4] значение анализа предикатов для проведения оптимизирующих преобразований резко возросло. Это связано с тем, что в этих архитектурах есть явные механизмы перевода потока управления в поток данных, а именно, поддержка предикатного режима исполнения операций. В зависимости от значения предиката, приписываемого операции, операция либо выполняется, либо нет. При переводе программы в предикатную форму часть операций передачи управления удаляется, а операции ставятся под соответствующие предикаты, обеспечивающие корректную семантику программы. Основная идея перевода потока управления в поток данных описана в [5]. В этой работе описана базовая техника построения корректного предиката для операций при удалении из программы части операций передачи управления.

Помимо чисто технической стороны перевода потока управления в поток данных, существует проблема выбора тех регионов программы, которые нужно переводить в предикатный вид. Это связано с тем, что в некоторых случаях перевод потока управления в поток данных может не дать никакой выгоды с точки зрения эффективности полученного кода. Для того, чтобы этот перевод давал реальную выгоду, в архитектурах VLIW и *EPIC* реализованы механизмы, позволяющие исполнять несколько элементарных операций в одном машинном такте. Для решения проблемы эффективного использования механизмов распараллеливания на уровне инструкций разработано несколько различных алгоритмов планирования, которые призваны решить проблему эффективного использования ресурсов широкой команды [6,7,8,9,10,11].

В основном эти алгоритмы отличаются по структуре регионов программы, которые рассматриваются в качестве кандидатов на преобразование в предикатный вид. В работах [6,7,8] рассматриваются ациклические регионы, а в работах [9,10,11] рассматриваются регионы с произвольным управлением. В контексте программы, преобразованной в предикатную форму, результаты анализа предикатов используются для решения нескольких задач. Наиболее известными являются задача определения отношения зависимости между операциями, задача определения времен жизни переменных [12,13], задача минимизации числа предикатов и др.

В данной работе мы вкратце рассмотрим основные подходы к анализу предикатов, предложим метод распространения анализа предикатов на случай произвольного управления. Также будут предложены несколько оптимизаций, работающих на основе анализа предикатов, и рассмотрен подход, позволяющий использовать результаты

анализа предикатов для решения некоторых проблем отладчика оптимизированных кодов.

1.Краткое описание существующих алгоритмов анализа предикатов

Можно выделить 2 различных подхода к анализу предикатов, которые широко используются в оптимизирующих компиляторах. Первый из них основан на построении Графа Разделения Предикатов (Predicate Partition Graph (PPG)) [14]. Для дальнейшего изложения нам понадобится следующее

Определение 1:

Узел управляющего графа *w* зависит по управлению от дуги управляющего графа (*u* **à** *v*) если выполняются следующие условия:

1. *w* постдоминирует [12,13] *v* и
2. если *w* не совпадает с *u*, тогда *w* не постдоминирует *u*.

Итак, PPG (predicate partition graph) это топология, которая связывает классы линейных участков в соответствии с возможностью их одновременного выполнения. Для построения PPG все узлы управляющего графа программы [12,13] разбиваются на классы эквивалентности по управлению.

Определение 2: два узла называются эквивалентными по управлению, если множества дуг, от которых они зависят по управлению, совпадает.

Каждому классу ставится в соответствие узел PPG. Далее, за один проход по дугам управляющего графа строятся дуги PPG. По построенному графу (PPG) строится бинарная матрица Disjoint размера n x n, где n - число узлов PPG. Матрица предназначена для быстрого ответа на вопрос о пересечении предикатов. Для ответов на другие вопросы об отношениях предикатов надо исследовать топологию PPG.

Другой подход основан на использовании Binary Decision Diagram (BDD) [15,16,17]. При использовании этого подхода для всех операций некоторого региона, переведенного в предикатную форму, которые вырабатывают предикаты, строится булева функция, переменными которой являются операции сравнения (CMP). Здесь следует отметить что, как правило, в программе, не переведенной в предикатную форму, предикатные значения вырабатывают только операции сравнения, которые являются аргументами операций условного перехода. Построенная функция представляется в виде выражения BDD, которое затем можно сравнивать с другими подобными выражениями над предикатами.

В дальнейшем везде подразумевается анализ предикатов на основе BDD.

2. Распространение анализа предикатов за пределы ациклических регионов.

Для того, чтобы анализ предикатов был более полон, необходимо уметь сопоставлять предикатные выражения не только предикатным операциям, но и φ-функциям [18]. φ-функции отражают места схождений потока данных в местах схождения потока управления. Каждому аргументу φ-функции ставится в соответствие дуга управляющего графа, входящая в узел управляющего графа, для которого проводится анализ. Если управляющий поток проходит по некоторой дуге e_i, то результатом φ-функции является её аргумент, которому сопоставлена дуга e_i. Сопоставим каждой дуге управляющего графа e_i переменную x_i^e, i = 1,..,n, где n – число входящих управляющих дуг. Рассмотрим некоторую φ-функцию $\varphi(arg_1,\ldots, arg_n)$. Пусть $p_1,\ldots, p_n$ - предикаты, сопоставленные аргументам $arg_1,\ldots, arg_n$. Тогда поставим в соответствие φ-функции φ предикат $p(\varphi)=\sum_{i=1}^{n} p_i x_i^e$, а в качестве отрицания предиката p(φ) рассмотрим

$\overline{p}(\varphi)=\sum_{i=1}^{n}\overline{p_i}x_i^e$, где под суммой понимается дизъюнкция. Таким образом, всякой предикатной операции и φ-функции поставлен в соответствие некоторый предикат.

Утверждение 1. Если для двух предикатных операций $oper_1$ и $oper_2$ выполнено $p(oper_1) \rightarrow p(oper_2)$, то $oper_1 \rightarrow oper_2$; если $p(oper_1) = p(oper_2)$, то $oper_1 = oper_2$.

Действительно, в случае отсутствия φ-функций в дереве аргументов каждой из операций, утверждение истинно, так как по построению p(oper)=oper. В случае φ-функции, необходимо проверить истинность формул $oper_1 \rightarrow oper_2$, $oper_1 = oper_2$ для любого управления, т.е. для любого предшественника φ-функции. Так как формулы $p(oper_1) \rightarrow p(oper_2)$, $p(oper_1) = p(oper_2)$ истинны на любом наборе переменных, следовательно, они истинны и на наборах x_1^e,…, x_n^e = (1,0,…,0),…,(0,…,0,1), на которых формулы принимают вид, где вместо φ-функций подставлены аргументы, соответствующие одному из возможных вариантов управления, что и требовалось проверить.

3. Использование результатов анализа предикатов в оптимизирующих компиляторах

3.1 Минимизация и упрощение предикатных выражений

Оптимизация применяется к предикатным операциям. Для предикатной операции oper рассматривается её предикатное представление p(oper). По этому представлению строится совершенная дизъюнктивная нормальная форма (СДНФ). Построенная СДНФ сначала упрощается путем исключения из ее конъюнкций логически излишних операций сравнений (переменных) (например, x>5 влечет x>3), и удаления тождественно ложных конъюнкций, содержащих логически противоречащие сравнения (например, условие x>2 противоречит условию x<0). Затем полученная СДНФ приводится к сокращенной СДНФ [15,16,17]. Если сокращенная СДНФ имеет меньшую длину в смысле количества операций конъюнкций и дизъюнкций, результат операции меняется на вычисления сокращённой СДНФ. В оптимизации не используется представления φ - функций в виде предикатных выражений, так как это требует введения новых переменных, не имеющих конкретной привязки к операционной семантике.

3.2 Удаление избыточных операций чтения из памяти на предикатном коде.

Для применимости оптимизации удаления избыточных чтений из памяти необходимо выполнимость следующего условия: операция записи, от которой берётся значение либо доминирует операцию чтения, либо доминирует все использования операции чтения. На не предикатном представлении эта информация получается из анализа дерева доминаторов [12,13]. На предикатном коде анализ на основе дерева доминаторов слишком консервативен, так как целые регионы теперь представляются в виде одного узла управляющего графа. Поэтому для анализа доминирования одной условно исполняемой операции над другой, можно использовать анализ предикатных выражений. Действительно, операция $oper_1[P_1]$ доминирует операцию $oper_2[P_2]$ тогда и только тогда, когда $P_2 \rightarrow P_1$ тождественная истина, где P_1 и P_2 - результаты некоторых предикатных операций $oper\_P_1$ и $oper\_P_2$. В силу утверждения 1 для факта доминирования операцией $oper_1[P_1]$ операции $oper_2[P_2]$ достаточно, чтобы предикатное

выражение $p(oper\_P_2) \rightarrow p(oper\_P_1)$ было тождественной истиной. На рисунке 1 представлен пример, где на основе анализа предикатов удаётся определить, что операция store a, v [p0] доминирует над операцией use r0 [p4], следовательно, оптимизация удаления избыточных чтений применима.

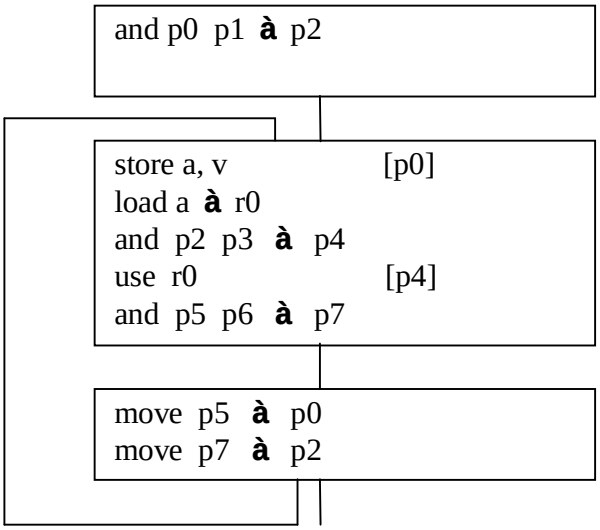

Рис.1: Пример предикатного кода

3.3 Удаление избыточных операций записи в память

Оптимизация состоит в объединении операций записи в память по одному и тому же адресу, стоящих под предикатами. Записываемое значение для новой операции записи берётся от второй операции в случае истинности предиката второй операции записи, иначе берётся записываемое значение первой операции. Новая операция записи ставится под объединение предикатов первой и второй операции. Пример применимости оптимизации показан на рисунке 2.

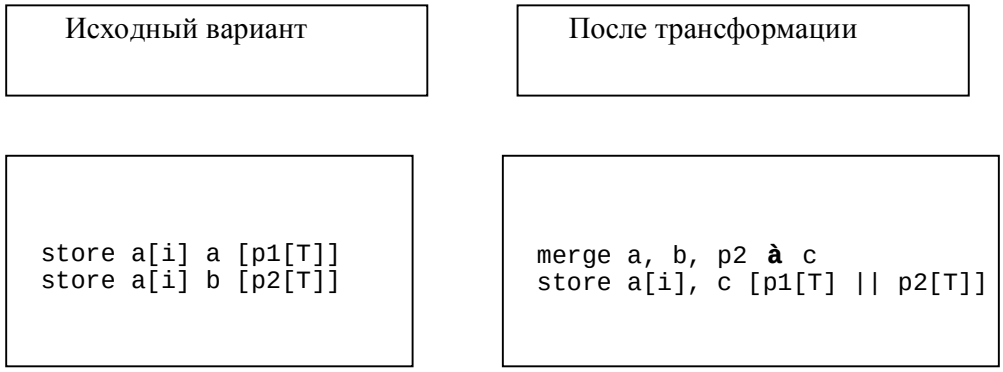

Рис. 2. Пример применимости оптимизации, удаляющей избыточные операции записи в память.

Используя анализ предикатных выражений в некоторых случаях можно повысить эффективность этой оптимизации. Например, если дизъюнкция предикатов операций записи есть тождественная истина, нет необходимости строить предикат для новой операции записи, а следует поставить её в безусловный режим исполнения. Если из выполнимости предиката первой операции записи следует выполнимость предиката второй операции записи, это означает, что вторая операция записи постдоминирует первую, и следует просто удалить первую операцию записи.

3.4 Удаление избыточных операций, спланированных после операции условного перехода

Оптимизация заключается в удалении ненужных предикатных операций, стоящих в узле графа управления после операции перехода. Суть оптимизации заключается в использовании того факта, что предикат операции перехода является тождественно ложным в любой точке узла управляющего графа, находящейся ниже этого перехода. На рисунке 2 приведён пример срабатывания оптимизации.

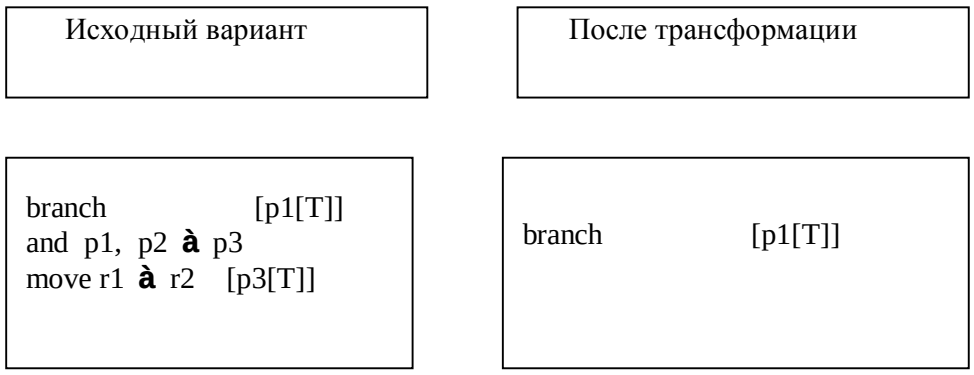

Рис. 3. Пример применимости оптимизации, удаляющей избыточные предикатные вычисления на основе анализа предикатов переходов.

3.5 Сбор отладочной информации на предикатном коде.

Сбор отладочной информации [19,20,21] при высоких уровнях оптимизации компилятора для VLIW/EPIC - архитектур приводит к необходимости его проведения на предикатном коде.

Результатом сбора отладочной информации является список точек останова – пар, где первым элементом является адрес широкой команды, вторым - номер строки текста исходной программы. Элементом списка пар является список пар <переменная, выражение>, в котором перечисляются все переменные, видимые в текущей точке останова. Каждой переменной соответствует выражение, которое в общем случае является деревом, узлами которого являются операции, а листьями - либо константы, либо адреса памяти, либо номера регистров. Через это выражение можно получить значение переменной в текущей точке останова.

Рассмотрим фрагмент программы, изображённый на рисунке 4, где цифрами обозначены номера строк исходного текста.

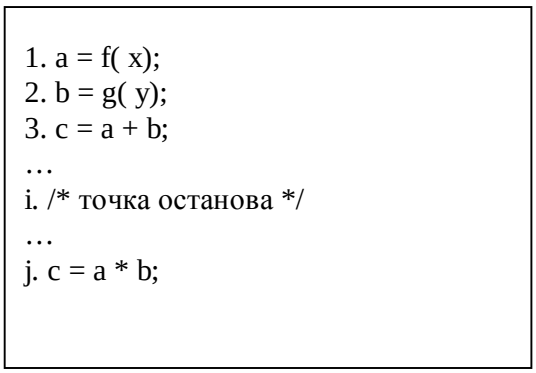

Рис. 4. Пример программы, содержащей избыточные вычисления.

Предположим, что между операциями 3 и j нет использований переменной c . В таком случае операций 3 является мертвым кодом и будет удалена. Однако переменные a и b живут от точки 1 и 2 до точки j, следовательно, значение переменной c, которой уже не соответствует ни ячейка памяти, ни регистр, может быть получена с помощью выражения через переменные a и b.

Рассмотрим пример с предикатным кодом, приведённый на рисунке 5.

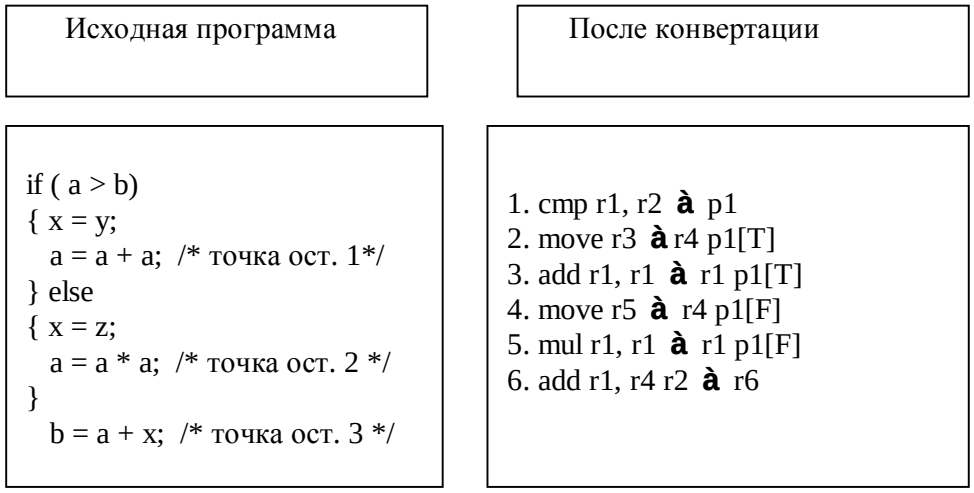

Рис. 5. Пример программы, переведённой в предикатный код.

На рисунке 5 операции упорядочены в соответствии с их номерами широких команд. Таким образом, первой точке останова соответствует широкая команда 3, второй – широкая команда 5, третьей – широкая команда 6. Так как присваивание в переменную x (регистр r4) осуществлялось в широкой команде 2 условно (под предикатом p1[T]), то в первой точке останова отладчик в ответ на запрос распечатать значение переменной x вынужден ответить, что значение переменной в данной точке недоступно(если только ранее уже не было известно, что значение переменной x лежит на регистре r4). В широкой команде 4 происходит условное присваивание в переменную x (регистр r4). Но теперь условие, которое гарантирует тот факт, что значение переменной x лежит на регистре r4 равно p1[F] || p1[T] = TRUE. Таким образом, во второй точке останова, отладчик сможет правильно распечатать значение переменной x. Заметим, что даже если в процессе исполнения программы p1 = TRUE, то отладчик остановится сначала в точке останова 1, в которой он не сможет гарантированно выдать значение переменной x. Далее отладчик перейдет не к точке останова 3, а к точке останова 2 и только там выдаст значение переменной x = y. Затем отладчик перейдет к точке останова 3 и там он также правильно распечатает значение переменной x. Таким образом, после преобразования управления в предикатное представление некоторые присваивания дополнительно снабжаются предикатами и для правильной распечатки значения переменной в некоторой точке останова необходимо проверить, что выражение, через которое доступно в текущей точке останова значение переменной снабжено предикатом, тождественно равным истине. Такая проверка осуществляется с применением анализа предикатных выражений.

Заключение

В работе рассмотрены методы использования результатов анализа предикатов для целей оптимизации программ. Предложен механизм, позволяющий обобщить анализ предикатов на случай регионов общего вида и предложены новые оптимизации, основанные на применении результатов обобщенного анализа. Также был предложен метод использования результатов анализа предикатов для решения проблемы отладчика оптимизированных кодов в контексте преобразования if-conversion. Все предложенные методы были реализованы в оптимизирующем компиляторе проекта «Эльбрус» [22,23] и показали высокую эффективность для решения рассмотренных задач оптимизирующей компиляции для архитектур с явно выраженным параллелизмом.

литература

1. **L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante**, “Predicated single static assignment “, in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, October 1999.
2. **Intel Itanium** 2 Processor Reference Manual, Document Number: 251110-001, June 2002.
3. **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.
4. **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.
5. **J.R. Allen, K. Kennedy, C. Porterfield, J. Warren**. Conversion of control dependences to data dependences, Conf. Record of POPL-10, 1983.

6. **Joseph A. Fisher.** Trace Scheduling: A technique for global microcode compaction // Transactions on Computers, IEEE. July, 1981. V. C-30. P. 478-490.
7. **S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann.** Effective Compiler Support for Predicated Execution Using the Hyperblock. // in Proceedings of the 25th International Symposium on Microarchitecture. December, 1992. P. 45-54.
8. **W. Hwu et al**., The superblock: an effective technique for VLIW and superscalar compilation, The Journal of Supercomputing, 1993, 229-248.
9. **Alexandru Nicolau and Steven Novack**, Trailblazing: A Hierarchical Approach to Percolation Scheduling. Department of Information and Computer Science University of California, Proceedings of the 1993 International Conference on Parallel processing.
10. **Steven Novack and Alexandru Nicolau**, An Efficient Global Resource Constrained Technique for Exploiting Instruction Level Parallelism. Department of Information and Computer Science University of California, Proceedings of the 1992 International Conference on Parallel Processing.
11. **Novack and Alexandru Nicolau**, A Hierarchical Approach to Instruction-level Parallelization. Department of Information and Computer Science University of California, International Journal of Parallel Programming, 23(1), 1995.
12. **Steven S. Muchnick**. Advanced Compiler Design and Implementation – Morgan Kauffman, San Francisco, 1997, chapter 7.2.
13. **Касьянов В.Н., Евстигнеев В.А.** Графы в программировании: обработка, визуализация и применение. – СПб.: БХВ-Петербург, 2003.
14. **Richard Johnson and Michael Schlansker**, "Analysis Techniques for Predicated Code", In Proc. Of the 29th Annual Int'l Symp. On Microarchitecture, December 1996.
15. **John Wollenburg**, "Condition awareness support for predicate analysis optimization ", University of Illinois, 1997.
16. **R.E. Bryant**. Symbolic Boolean manipulation with ordered binary decision diagrams. Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1992.
17. **D.I. August, J.W. Sias, J. Puiatti, S.A. Mahlke, D.A. Connors, K.M. Crozier, and W.W. Hwu,** "The program decision logic approach to predicated execution", in Proceedings of the 26th International Symposium on Computer Architecture, pp. 208-219, May 1999.
18. **J. Ferrante, K. J. Ottenstein, and J.D. Warren**. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3):319-349, July 1987.
19. **Le-Chun Wu, Wen-mei W. Hwu**. A New Data-Location Tracking Scheme for the Recovery of Expected Variables Values. Technical Report IMPACT-98-07
20. **Le-Chun Wu, Rajiv Mirani, Harish Patil, Bruce Olsen, Wen-mei W. Hwu**. A New Framework for Debugging Globally Optimized Code. ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia, May 1-4, 1999.
21. **R. Gupta**, "Debugging code reorganized by a trace scheduling compiler", Structred Programming, vol. 11, pp. 141-150, July 1990.
22. **М. Кузьминский**. Отечественные микропроцессоры: Elbrus E2K // Открытые системы, № 05-06, 1999. – С. 8-13.
23. **K. Dieffendorf**. The Russians Are Coming. Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee // Microprocessor Report, V.13, ¹.2. – February 15, 1999. - P. 1-7.