Модел. и анализ информ. систем. Т. **20**, № **2** (2013) 166–177

УДК 004.451.2

Построение универсального линеаризованного графа потока управления для использования в статическом анализе кода алгоритмов

Битнер В.А., Заборовский Н.В.

Московский физико-технический институт (государственный университет)
141700, Московская область, г. Долгопрудный, Институтский переулок, 9

e-mail: vbitner87@gmail.com nzaborovsky@parallels.com

получена 25 марта 2013

Ключевые слова: состояние гонки, статический анализ, многопоточные алгоритмы, SSA, оптимизирующий компилятор

В работе рассматривается вариант построения универсального линеаризованного графа потока управления, архитектурно-независимого и пригодного для описания программы любого языка программирования высокого уровня. Практическая польза данного графа заключается в возможности быстрого и оптимального поиска уникальных путей исполнения, что может быть особенно ценно в методах статического анализа кода алгоритмов с целью поиска в них состояния гонки («race condition»). В качестве технического средства для построения линеаризованного графа управления используется оптимизирующий компилятор CLANG&LLVM. В работе проводится анализ попроцедурных оптимизаций компилятора LLVM, трансформация промежуточного представления которых приводит как к сокращению количества инструкций условного и безусловного перехода в коде, так и к удалению или упрощению целых циклов и условных конструкций. Результат анализа, приведенный в работе, позволил выявить наиболее эффективную линейку оптимизаций компилятора LLVM, которая приводит к существенной линеаризации графа потока управления, что было продемонстрировано на примере кода взаимоисключающего алгоритма Петерсона для 2 потоков.

1. Введение

Проведя анализ современных оптимизирующих компиляторов таких компаний, как Intel, Sun Microsystems, Transmeta, Microsoft, IBM, HP, Elbrus [1], а также особо уделяя внимание компиляторам с открытым кодом, таким как CLANG&LLVM, GCC, понимаешь, что задача линеаризации графа потока управления для организации более эффективного статического анализа алгоритмов имеет пересечения с задачами оптимизирующих компиляторов – получения эффективного кода целевой архитектуры. Особенно данный факт проявляется для оптимизирующих компиляторов архитектурных платформ, использующих статический подход к распараллеливанию на уровне инструкций – EPIC-архитектуры (Explicitly Parallel Instruction

Computing), а также архитектурных платформ с явной поддержкой параллелизма на уровне отдельных инструкций с широким командным словом – VLIW-архитектуры (Very Long Instruction Word).

EPIC- и VLIW-архитектуры, как правило, имеют длинный конвейер, из-за чего остро стоит проблема условных переходов, т.к. в момент исполнения операций условного перехода процессору неизвестно до конца, какая из веток условного перехода будет исполняться. В зависимости от успешности предсказаний условного перехода процессор либо исполняет ранее подкачанные инструкции из конвейера, либо освобождает конвейер и загружает нужные инструкции, что, очевидно, негативно сказывается на производительности исполнения программ. Для решения данной проблемы немалый вклад вносит оптимизирующий компилятор той или иной платформы, который осуществляет различные оптимизации на промежуточном представлении (IR – intermediate representation) программ в процессе их компиляции.

Процесс оптимизации промежуточного представления компилятора разделен на фазы оптимизации, где каждая фаза реализует отдельную, чаще независимую оптимизацию над IR [2]. Все фазы оптимизации формируют линейку оптимизаций, которая в зависимости от реализации компилятора может меняться по контексту IR либо по требованию пользователя. Особенность реализации компиляторов позволяет получать IR практически после каждой фазы оптимизации, а также соответствующие абстрактные представления этого IR: граф потока управления (control flow graph – CFG), граф потока данных (def-use graph – DUG), дерево доминаторов и т.д.

Таким образом, есть возможность использовать и адаптировать некоторые оптимизации компилятора с целью получения наиболее удобного и эффективного представления программы, написанной на языке высокого уровня, для проведения статического анализа алгоритмов на предмет состояния гонки.

2. Общие понятия и постановка задачи

Введем несколько определений, связанных с предметной областью, и поясним, какую значимость имеют линеаризованные представления в статическом анализе программы с целью выявления состояния гонок.

Состояние гонки (race condition) – ситуация, когда несколько потоков одновременно обращаются к одному и тому же ресурсу, причем хотя бы один из потоков выполняет операцию записи, и порядок этих обращений точно не определен. В случае реализации состояния гонок исполнение программы может привести к недетерминированным результатам.

Методики поиска состояний гонок обычно разделяют на *статический анализ*, *динамический анализ*, *проверку на основе моделей и аналитические доказательства корректности программ*. Наиболее полный обзор данных методик и программных технологий, использующих подобные методики, представлен в работе [12]. В работе уделяется внимание *статическому анализу* – анализу кода без исполнения программы. Интерес представляет статический анализ *неблокирующих алгоритмов (lock-free)* – класс многопоточных алгоритмов, в которых для синхронизации не используются механизмы блокировки потоков исполнения.

Существуют различные методы статического анализа [12], которые обладают

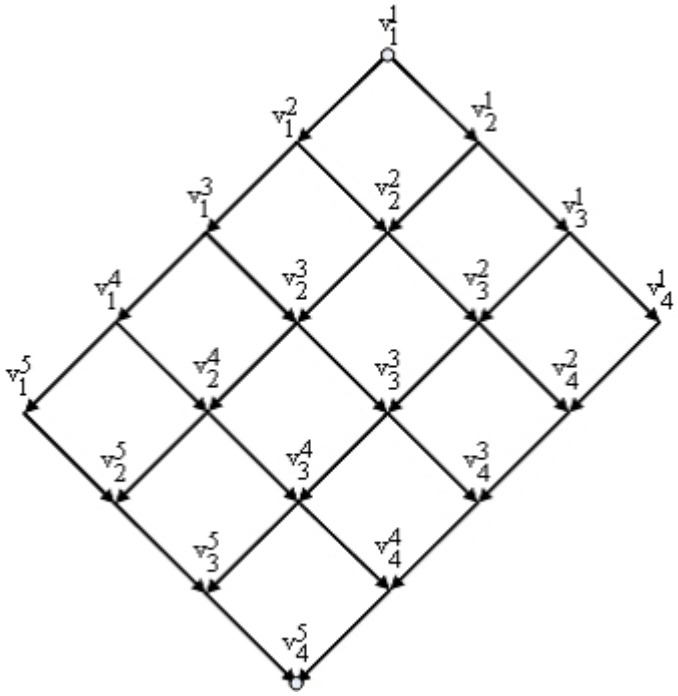

Рис. 1. Пример графа совместного исполнения потоков при $k = 4$, $n = 3$

как преимуществами, так и недостатками, но в рамках работы интерес представляет метод, который основывается на графах совместного исполнения потоков и расчетного графа [3, 12].

Граф совместного исполнения потоков – ориентированный граф, представляющий всевозможные варианты совместного исполнения потоков в многопоточном алгоритме, где каждая дуга ассоциирована с атомарной операцией одного из потоков, а вершины – с множеством состояний общей памяти после выполнения очередной атомарной операции. В случае двух потоков исполнения граф можно описать следующим образом:

$$G := (V, A) : V = \bigcup_{\substack{i=1,k+1\\ j=1,n+1}} v_j^i, A = \bigcup_{\substack{i=1,k\\ j=1,n+1}} (v_j^i, v_j^{i+1}) \cup \bigcup_{\substack{i=1,k+1\\ j=1,n}} (v_j^i, v_{j+1}^i),$$

где k, n – количество операций первого и второго потока соответственно, i, j – номер атомарной операции первого и второго потока соответственно, V – множество вершин графа, A – множество дуг графа. Пример подобного графа представлен на рис. 1.

Расчетный граф – конструктивная дискретная модель исходного кода программы, представляющая собой направленный граф, в котором каждому ребру соответствует атомарная операция, а вершинам ставится в соответствие множество значений всех разделяемых переменных.

Общая идея подхода к задаче о нахождении состояния гонки на основе графа совместного исполнения потоков и расчетного графа заключается в следующем [3]:

1. Находятся классы эквивалентности полных путей на графе совместного исполнения потоков.

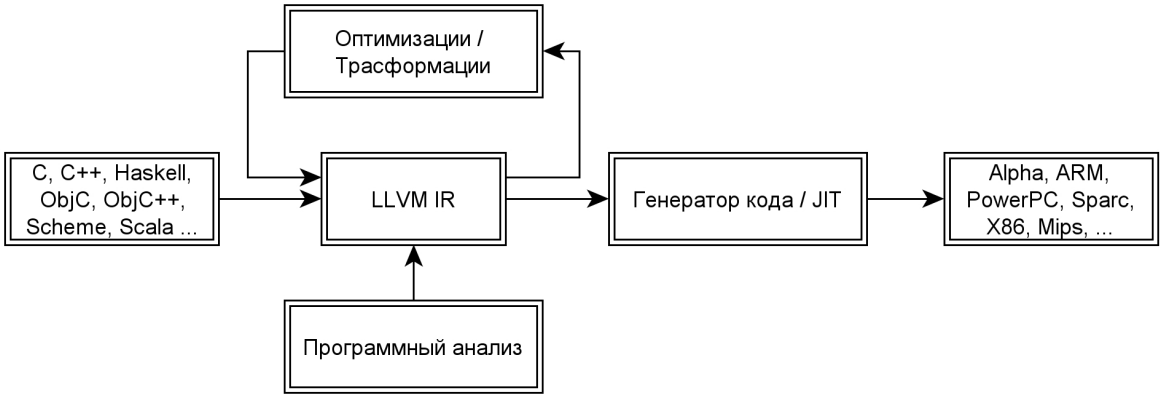

Рис. 2. Структура компиляции с помощью CLANG&LLVM

2. Анализируются полные пути на графе совместного исполнения потоков, которые принадлежат разным классам эквивалентности. На основе анализа выбираются пути, для которых возможно состояние гонки.

3. На расчетном графе анализируются только пути, выбранные в п. 2, и выводится результат о наличии или отсутствии гонки.

Подобный метод статического анализа, так же как и многие другие, чувствителен к условным переходам и циклам в программах. Усложняется не только анализ, но и появляется риск неприменимости анализа. Таким образом, линеаризация представления программы, на котором строится граф совместного исполнения потоков и расчетный граф, позволяет расширить класс задач, к которым можно применить статический анализ.

В работе ставится задача разработать алгоритм и технологию получения представления программы, которое содержало бы наименьшее количество условных и цикловых конструкций. Как следствие, возникнет возможность строить линеаризованный и наиболее удобный для статического анализа CFG программ.

3. Анализ оптимизаций на промежуточном представлении

В нашей работе был взят за основу компилятор CLANG&LLVM как наиболее активно развивающийся и обладающий рядом преимуществ с точки зрения гибкости реализации и удобства адаптирования под конкретные случаи и задачи. CLANG&LLVM для проведения оптимизаций использует платформонезависимый IR в SSA (Static Single Assignment) форме, причем CLANG выступает front-end'ом, транслируя язык высокого уровня в IR нужного вида (LLVM IR), а LLVM – оптимизирующим компилятором, который проводит необходимые оптимизации, трансформации или программный анализ на LLVM IR и затем транслирует финальный LLVM IR в бинарный код целевой архитектуры. Общая структура использования компилятора представлена на рис. 2.

Предлагается рассмотреть и использовать следующие оптимизации, которые часто имеют свою реализацию в оптимизирующих компиляторах:

- If-conversions – перевод в предикатное представление [4]. Современные EPIC-архитектуры поддерживают предикатное вычисление, что позволяет выполнять операции раньше перехода, ведущего на соответствующий участок. Вынесенные операции выше перехода исполняются под условием (предикатом) этого перехода. Используя поддержку предикатных вычислений, можно преобразовать ациклический регион программы, состоящий из нескольких линейных участков, в один линейный участок. При этом действии будут удалены все внутренние переходы ациклического региона. По сути, if-conversion преобразует зависимости по управлению с операциями переходов в зависимости по данным с предикатами. В рамках выбранного оптимизирующего компилятора CLANG&LLVM преобразование промежуточного представления в предикатную форму не реализовано, но предоставлен соответствующий API [5], который позволяет оперировать с нужными структурами IR и его представлениями. Таким образом, возможна реализация if-conversion, основанная на известных алгоритмах преобразования и общепринятых практиках [4, 7].

- Трансформация циклов: В современных оптимизирующих компиляторах реализованы десятки или даже сотни различных трансформаций циклов, некоторые из них можно отнести к оптимизациям, применение которых неявно приводит к линеаризации CFG. Эти и множество других трансформаций циклов подробно описаны в обзорной статье [8]. В рамках данной работы рассмотрены такие оптимизации CLANG&LLVM, применение которых с практической точки зрения наилучшим образом отвечают задаче линеаризации CFG. Согласно официальной документации оптимизирующего компилятора LLVM [9] возможно применить следующие оптимизации к циклам на IR, полученные в данном случае с помощью CLANG:

  – *-lcssa (Loop-Closed SSA Form Pass)*
  Оптимизация изменяет циклы путем переноса фи-узлов в конец циклов для всех значений, которые остаются живыми за границами цикла. Например:

```
for (...)                    for (...)
    if (c)                       if (c)
        X1 = ...                     X1 = ...
    else                 ->      else
        X2 = ...                     X2 = ...
    X3 = phi(X1, X2)             X3 = phi(X1, X2)
... = X3 + 4                  X4 = phi(X3)
                                 ... = X4 + 4
```

  Дополнительные фи-узлы являются излишними, но они легко удалятся после оптимизации InstCombine. Основная выгода данной трансформации – это сделать применение других цикловых оптимизаций, таких как LoopUnswitch, более простыми.

  – *-licm (Loop Invariant Code Motion)*
  Оптимизация выполняет перемещение инвариантного кода в цикле, пытаясь удалить как можно больше кода из тела цикла.

– *-loop-deletion (Delete dead loops)*
Оптимизация отвечает за удаление циклов, которые не вносят вклад в возвращаемое значение функции.

– *-loop-reduce (Loop Strength Reduction)*
Оптимизация проводит понижение стоимости выражений с индуктивными переменными, заменяя дорогие операции на более дешевые. В частности выражения с массивами изменяют так, чтобы воспользоваться преимуществами расширенного режима адресации индекса (scaled-index addressing modes), упрощающего обращение к элементам массивов.

– *-loop-rotate (Rotate Loops)*
Оптимизация выполняет простое вращение цикла.

– *-loop-simplify (Canonicalize natural loops)*
Оптимизация выполняет несколько трансформаций над естественными циклами, преобразуя их в более простую форму, которая позволяет делать последующий анализ и трансформации проще и эффективнее, например, для LICM (Loop Invariant Code Motion). Данная оптимизация создает дополнительный узел (предцикл) перед головой цикла, что гарантирует наличие только единственной входной дуги в цикл. Также гарантируется, что измененный цикл будет иметь только одну обратную дугу.

– *-loop-unroll (Unroll loops)*
Оптимизация реализует простую раскрутку цикла, которая заключается в создании нескольких копий итераций цикла (число копий n называет фактором развертки) и в увеличении шага цикла в это же число раз. Например, развертка с фактором 2:

```
for (i = 0; i < n; i++)          for (i = 0; i < n-1; i+=2) {
    a[i] = b[i] + c[i];              a[i] = b[i] + c[i];
                              ->     a[i+1] = b[i+1] + c[i+1];
                                 }
                                 for (; i < n; i++)
                                     a[i] = b[i] + c[i];
```

Оптимизацию стоит применять после исполнения оптимизации -indvars (Canonicalize Induction Variables), когда циклы приведены к каноническому виду, что позволяет легко определять число итераций циклов.

– *-loop-unswitch (Unswitch loops)*
Оптимизация изменяет циклы, которые содержат переходы на инвариантные для цикла условные операции для того, чтобы создать несколько циклов. Например:

```
for (...)                        if (lic)
    A                                for (...)
    if (lic)          ->                 A; B; C
        B                        else
    C                                for (...)
                                         A; C
```

Ожидается, что оптимизация LICM исполняется раньше, что делает возможность применения данной оптимизации очевиднее.

– *-loop-idiom (Recognize loop idioms)*
Оптимизация реализует распознаватель идиом, который трансформирует простые циклы в бесцикловую форму.

– *-loop-instsimplify (Simplify instructions in loops)*
Оптимизация проводит легкое упрощение инструкций в теле цикла.

Среди представленных оптимизаций LLVM можно выбрать те, которые могут привести к существенному изменению CFG, а также прочих представлений IR, что в конечном итоге повышает линеаризацию CFG программ и упрощает различные виды статического анализа кода программ на предмет состояния гонок. Стоит заметить, что для успешной линеаризации CFG программ важно не только применить необходимые оптимизации компилятора LLVM, но необходимо правильно подобрать последовательность запусков оптимизаций, так как оптимизации могут быть конфликтующими – применимость одной приводит к неприменимости другой, и наоборот, связанными – неприменимость одной влечёт неприменимость другой [10].

Таким образом, наибольший интерес с точки зрения влияния на CFG вызывают следующие оптимизации:

- -lcssa,
- -loop-simplify,
- -licm,
- -loop-reduce,
- -loop-unroll,
- -loop-unswitch.

4. Применение оптимизаций LLVM

Компилятор позволяет создавать свою собственную линейку оптимизаций либо использовать стандартную. Стандартных линеек оптимизаций у LLVM 3: O1, O2, O3. Их также называют уровнями оптимизаций, где O1 – минимальный набор оптимизаций, а O3 – максимальный. Согласно документации LLVM интересуемый набор оптимизаций, выявленный в п. 3, содержится во втором уровне оптимизации LLVM – O2. Таким образом, достаточно использовать O2 во время компиляции, чтобы получить минимальный эффект от данных оптимизаций.

В качестве примера интересно рассмотреть CFG-графы взаимоисключающего алгоритма Петерсона для 2 потоков, реализация которого на языке Си выглядит следующим образом:

```
#define TRUE 1
#define FALSE 0
```

```
void enterRegion(int threadId)
{
    int other = 1 - threadId;       //Идентификатор второго потока
    interested[threadId] = TRUE;    //Индикатор интереса текущего потока
    turn = other;                   //Флаг очереди исполнения

    while (turn == other && interested[other]);
}

void leaveRegion(int threadId)
{
    interested[threadId] = FALSE;
}
```

Рассмотрим функцию enterRegion, трансляция которой в IR LLVM происходит с помощью CLANG, после чего применяются оптимизации на уровне LLVM. До применения оптимизаций CFG функции на IR LLVM можно увидеть на рис. 3. Как видно из рисунка, промежуточное представление функции приближено к ассемблерному представлению, но при этом типизировано. Несмотря на простую и короткую реализацию функции на языке высокого уровня, промежуточное представление enterRegion выглядит весьма внушительно, что уже может существенно усложнить процесс статического анализа: построение графа совместного исполнения потоков и расчетный граф, а также анализ на этих графах. Применение статического анализа в таком виде на реальных коммерческих программных продуктах с большой вероятностью будет затруднительным.

CFG функции enterRegion после применения необходимых оптимизаций LLVM можно увидеть на рис. 4. Количество инструкций IR LLVM и условных переходов в CFG функции существенно меньше. Эффект от применения оптимизаций на реальных задачах будет больше, что сделает применение статического анализа, описанного в работе [3], более доступным.

Статический анализ на графах совместного исполнения потоков и расчетном графе будет строиться на оптимизированном IR LLVM, где единицей анализа будет инструкция IR, приближенная к ассемблерному коду.

5. Заключение и дальнейшие планы работ

В работе был проведен анализ средств получения представления программ, которое удобно для построения линеаризованного графа потока управления для дальнейшего проведения статического анализа на предмет состояния гонок. Как результат данного анализа было принято решение использовать промежуточное представление оптимизирующего компилятора CLANG&LLVM, который обладает необходимыми свойствами для проведения архитектурно независимого статического анализа программ, написанных на языках высокого уровня.

На основе оптимизирующего компилятора CLANG&LLVM удалось получать линеаризованный CFG программ, который строится на оптимизированном промежу-

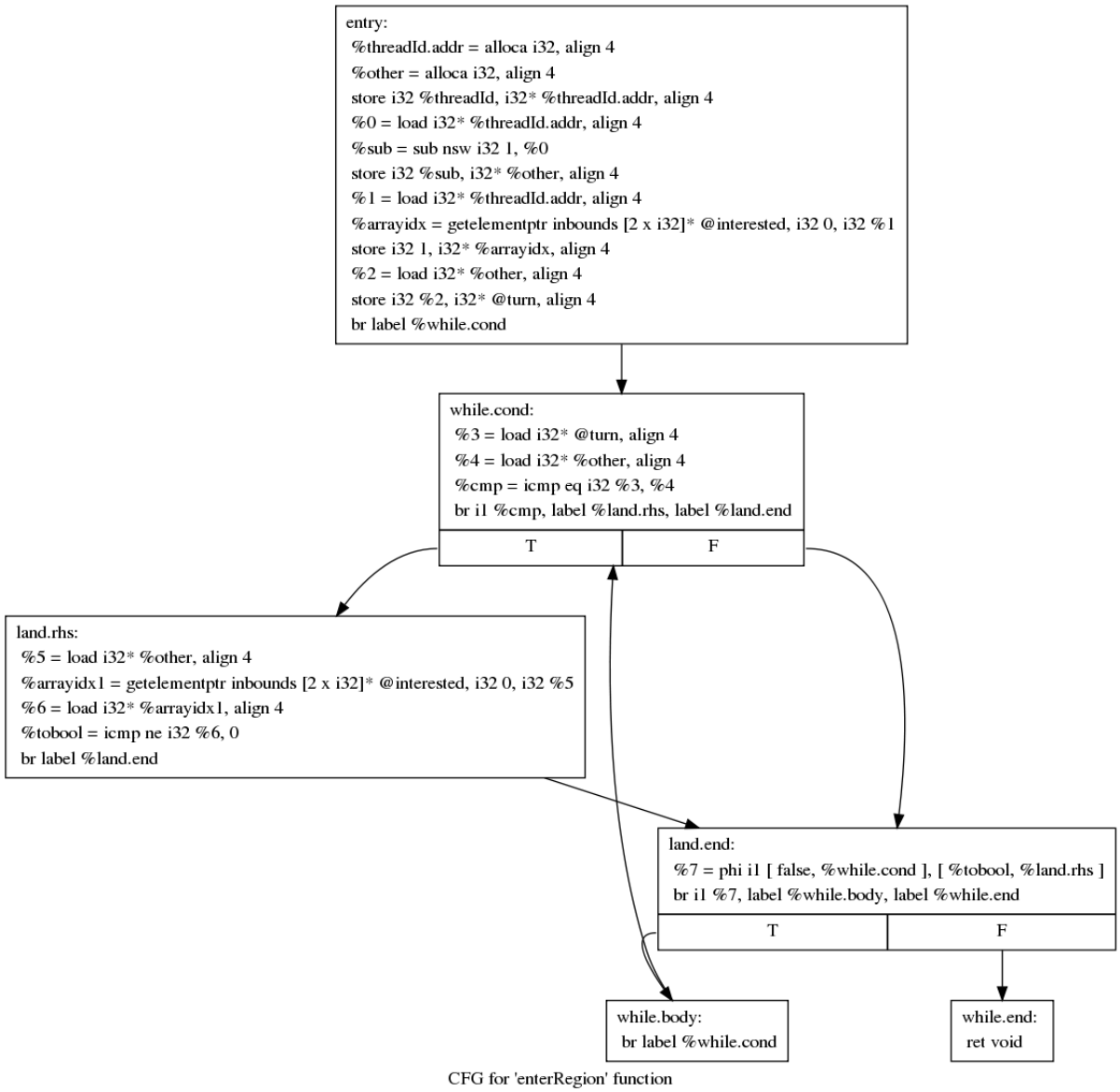

Рис. 3. CFG функции enterRegion до применения оптимизаций LLVM

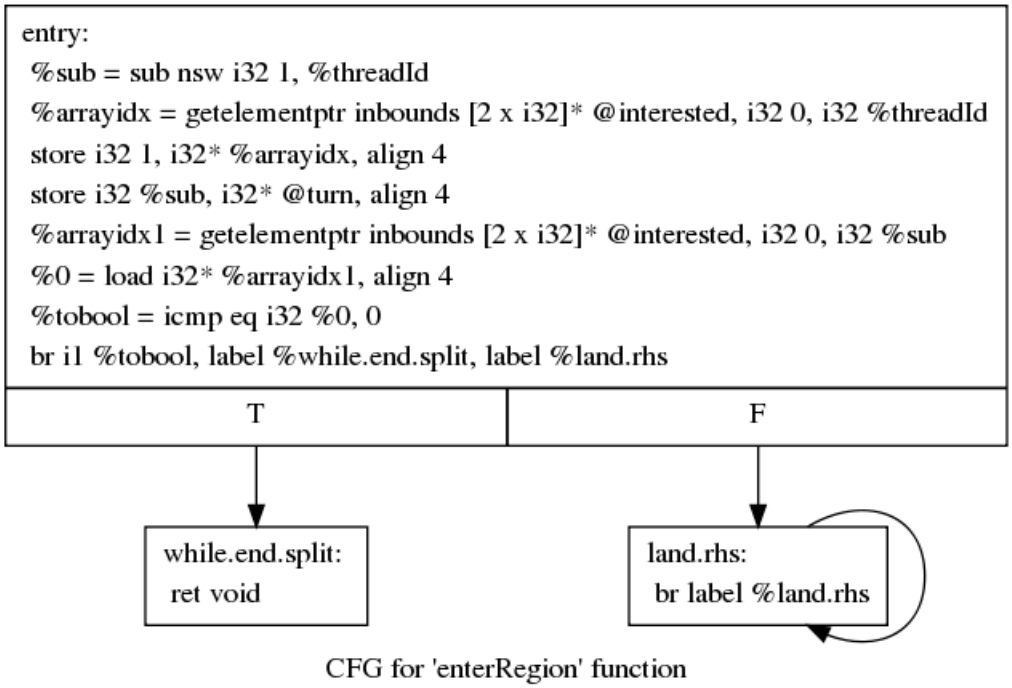

Рис. 4. CFG функции enterRegion после применения оптимизаций LLVM

точном представлении программ. Линеаризация CFG является следствием выявления необходимых оптимизаций из всего спектра оптимизаций LLVM, что составляет более 100 уникальных оптимизаций, и обязательного применения выявленных оптимизаций в процессе трансформации промежуточного представления программы. Таким образом, полученный вид CFG позволяет применить статический анализ [3] к более широкому классу реальных задач и сделать это эффективнее.

Дальнейшее изыскание в линеаризации CFG предполагает реализацию if-conversion для LLVM, что позволит еще больше избавиться от условных переходов в представлении. Также предполагается анализ графа совместного исполнения потоков и расчетного графа, построенный на новом линеаризованном представлении, с целью более детального изучения статического анализа [3] и более детальной оценки эффективности метода на реальных задачах.

Список литературы

1. Дроздов А.Ю. Компонентный подход к построению оптимизирующих компиляторов: автореф. дис. ... д-ра техн. наук: 05.13.11. М., 2010. 50 с. (Drozdov A.Yu. Komponentny podkhod k postroeniyu optimiziruyushchikh kompilyatorov: avtoref. dis. ... d-ra tekhn. nauk: 05.13.11. Moskva, 2010. 50 s. [in Russian]).

2. Ахо А.В., Сети Р., Ульман Д.Д. Компиляторы: принципы, технологии и инструменты. М.: Вильямс, 2003. 768 с. (English transl.: Compilers: Principles, Techniques, and Tools / Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, 1986.)

3. Заборовский Н.В. Расчетная модель нахождения состояний гонок в многопоточных алгоритмах: дис. ... канд. физ.-мат. наук: 05.13.18. М., 2011. 104 с. (Zaborovsky N.V.

Raschetnaya model nakhozhdeniya sostoyany gonok v mnogopotochnykh algoritmakh: dis. ... kand. fiz.-mat. nauk: 05.13.18. Moskva, 2011. 104 s. [in Russian]).

4. Дроздов А.Ю., Новиков С.В., Шилов В.В. Эффективный алгоритм преобразования потока управления в поток данных // Приложение к журналу “Информационные технологии”. 2005. № 2. С. 24–31 (Drozdov A.Yu., Novikov S.V., Shilov V.V. Effektivny algoritm preobrazovaniya potoka upravleniya v potok dannykh // Prilozhenie k zhurnalu "Informatsionnye tekhnologii". 2005. № 2. S. 24–31 [in Russian]).

5. LLVM API Documentation [Электронный ресурс]. Режим дост.: `http://llvm.org/docs/doxygen/html/IfConversion_8cpp.html`

6. Дроздов А.Ю., Новиков С.В. Исследование методов преобразования программы в предикатную форму для архитектур с явно выраженной параллельностью // Компьютеры в учебном процессе. 2005. № 5 (Drozdov A.Yu., Novikov SV. Issledovanie metodov preobrazovaniya programmy v predikatnuyu formu dlya arkhitektur s yavno vyrazhennoy parallelnostyu // Kompyutery v uchebnom protsesse. 2005. № 5 [in Russian]).

7. Muchnick Steven S. Advanced Compiler Design and Implementation, San Francisco: Morgan Kauffman, 1997.

8. Bacon D.F., Graham S.L., Sharp O.J. Compiler transformations for high performance computing // ACM Computing Surveys. 1994. 26(4). P. 345–420.

9. LLVM’s Analysis and Transform Passes [Электронный ресурс]. Режим дост.: `http://llvm.org/docs/Passes.html`

10. Дроздов А.Ю., Степаненков А.М. Методы комбинирования алгоритмов анализа и оптимизаций в современных оптимизирующих компиляторах // Компьютеры в учебном процессе. 2004. №11. С. 3–12 (Drozdov A.Yu., Stepanenkov A.M. Metody kombinirovaniya algoritmov analiza i optimizatsy v sovremennykh optimiziruyushchikh kompilyatorakh // Kompyutery v uchebnom protsesse. 2004. №11. S. 3–12 [in Russian]).

11. Медведев О.В. Линеаризация графа потока управления с учётом результатов профилирования // Системное программирование. 2006. Т. 2, №1. С. 25–47 (Medvedev O.V. Linearizatsiya grafa potoka upravleniya s uchetom rezultatov profilirovaniya // Sistemnoe programmirovanie. 2006. T. 2, № 1. S. 25–47 [in Russian]).

12. Кудрин М.Ю., Прокопенко А.С., Тормасов А.Г. Метод нахождения состояний гонки в потоках, работающих на разделяемой памяти // Сборник научных трудов МФТИ. М.: МФТИ, 2009. № 4. Том 1. С. 181–201 (Kudrin M.Yu., Prokopenko A.S., Tormasov A.G. Metod nakhozhdeniya sostoyany gonki v potokakh, rabotayushchikh na razdelyaemoy pamyati // Sbornik nauchnykh trudov MFTI. Moskva: MFTI, 2009. № 4. Tom 1. S. 181–201 [in Russian]).

The Construction of an Universal Linearized Control Flow Graph for Static Code Analysis of Algorithms

Bitner V.A., Zaborovsky N.V.

MIPT, Informatics Department
9 Institutskiy lane, Dolgoprudny city, Moscow Region, 141700, Russia

Keywords: race condition, static analysis, multithreaded algorithms, SSA, optimizing compiler

This paper presents the description of a possible way to build the universal linearized control flow graph which is supposed to be architecture-independent and applicable to the description of any high level language programs. The practical usefulness of the graph considered is the existence of the fast and optimal search of the unique execution paths that is valuable in the methods of static code analysis of algorithms for race condition search. Optimizing compiler CLANG&LLVM is used as a technical tool for building a linearized control flow graph. The analysis of LLVM compiler procedural optimizations is carried out in the article. Transformations of intermediate representation of those optimizations result in reduction of the number of instructions responsible for conditional or unconditional branches in the code as well as the elimination or simplification of the whole loops and conditional constructions. The results of the analysis performed in the paper allowed revealing the most effective optimizations line of the LLVM compiler, which leads to a significant linearization of the control flow graph. That fact was demonstrated by the example code of the Peterson mutual execution algorithm for 2 threads.

Сведения об авторах:
Битнер Вильгельм Александрович,
Московский физико-технический институт (государственный университет),
аспирант кафедры информатики;
Научно-технический центр ИБМ;
Заборовский Никита Владимирович,
Московский физико-технический институт (государственный университет),
лектор кафедры информатики;
ООО «Параллелз»