Федеральное государственное автономное образовательное учереждение
высшего образования
«Московский физико-технический институт (государственный
университет)»
Физтех-школа Радиотехники и Компьютерных Технологий
Кафедра микропроцессорных технологий в интеллектуальных системах
управления

Направление подготовки: 03.03.01 Прикладные математика и физика
Направленность (профиль): Радиотехника и компьютерные технологии

Анализ и улучшение алгоритма расположения инвариантов цикла в компиляторной инфраструктуре LLVM с использованием статического и динамического профилирования

бакалаврская работа

Студент:
Найданов Евгений Максимович

Научный руководитель:
Владимиров Константин Игоревич

Москва 2023

Аннотация

Анализ и улучшение алгоритма расположения инвариантов цикла в компиляторной инфраструктуре LLVM с использованием статического и динамического профилирования

Найданов Евгений Максимович

В работе рассматривается алгоритм расположения инвариантов цикла в компиляторной инфраструктуре LLVM. Продемонстрировано, что алгоритм неэффективен и требует улучшения. Предлагаются улучшения алгоритма и показывается, что улучшеный алгоритм обеспечивает оптимальное расположение инвариантов, обладает удовлетворительной ассимптотикой и увеличивает производительность компилируемых программ.

Оглавление

Введение

Расположение инвариантов цикла

Как известно, время исполнения программы распределено неравномерно по коду программы. Подавляющая часть этого времени, по некоторым оценкам порядка 90%, проводится в малой части кода программы (10% соответственно) [1]. Такие участки зачастую являются телом некоторого цикла.

Рассмотрим инварианты цикла - результат исполнения инструкции не обладающей сторонними эффектами, который не зависит от итерации цикла. Для значений существует множество расположений – базовых блоков графа потока управления, в которых расположены инструкции, порождающие эти значения. Вследствие свойств инвариантов цикла, для них мощность такого множества больше, чем для остальных переменных в цикле. Так, например, в отличие от индуктивной переменной, инвариант может быть вынесен из цикла.

Как уже было сказанно выше, скорость исполнения тела цикла является определяющим фактором скорости исполнения программы в целом. Из этого утверждения становится ясна важность минимизации времени исполнения блоков цикла. Субоптимальное расположение инвариантов цикла может негативно влиять на время исполнения программы, так как увеличит число инструкций, которое необходимо исполнить.

Мотивационный пример

На первый взгляд, достаточно вынести инварианты из цикла, расположив их перед входом в цикл [2]. Эта трансформация корректна, так как инвариант будет определен в точке его использования. Однако, такой подход не всегда обеспечивает минимальную суммарную частоту исполнения блоков расположения инварианта.

Примером случая, когда вынос инвариантов негативно влияет на производительность, является функция `S_regmatch` из бенчмарка `perlbench_r` который является частью набора бенчмарков SPEC CPU® 2017.

Данная функция имеет структуру, представленную на схеме 1.

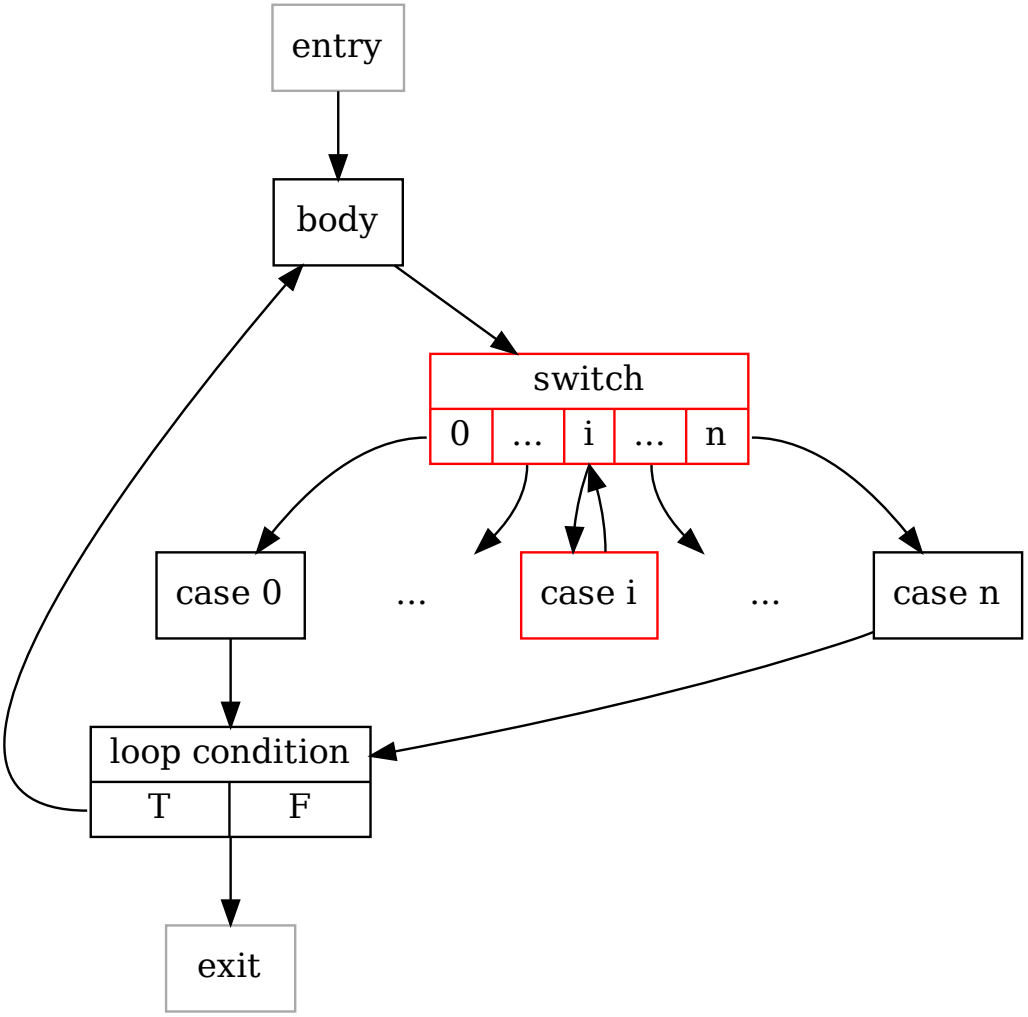

Рис. 1: Схема потока управления в функции `S_regmatch`

Функция состоит из внешнего цикла:
`body` → `switch` → `case ...` →`loop condition` → `body`

Внутри цикла находится конструкция `switch` с большим количеством вариантов. Большинство вариантов передают поток управления обратно к условию цикла, но некоторые передают его внутрь цикла, в базовый блок завершающийся конструкцией ветвления. Таким образом, образуется внутренний цикл:
`switch` → `case i` → `switch`

Вследствие данной структуры графа потока управления, в ходе исполнения программы тело внешнего цикла посещается чаще, чем тело внутреннего цикла. Таким образом, если некоторый инвариант внутреннего цикла, который не является инвариантом внешнего цикла, будет вынесен из внутреннего цикла, он окажется в теле внешнего цикла и число исполнений этой инструкции увеличится.

Постановка задачи

На приведенном выше примере, Clang (компилятор языка C, использующий компиляторную инфраструктуру LLVM), показывает субоптимальный результат. Среднее число инструкций на итерацию цикла больше, чем у компилятора GCC.

Это показывает на возможность улучшения алгоритма расположения инвариантов цикла в компиляторной инфраструктуре LLVM, что и является задачей данной работы.

Для решения этой задачи необходимо:

- Исследовать алгоритм расположения инвариантов цикла в компиляторной инфраструктуре LLVM.
- Разработать модификации алгоритма.
- Доказать, что модификации алгоритма приводят к более оптимальному расположению инвариантов цикла.
- Реализовать предложенные улучшения в компиляторной инфраструктуре LLVM.
- Продемонстрировать эффективность улучшений, измерив изменение производительности компилируемых програм.

Глава 1

Основные сведения из теории компиляторов

1.1 Циклы

Под циклом[3] в данной работе подразумевается максимальное подмножество вершин графа потока управления, такое что:

- Индуцированный подграф сильно связан.
- Все ребра, входящие в подмножество, направлены в единственную вершину - заголовок.

Для дальнейшего описания обработки циклов в компиляторной инфраструктуре, требуются следующие определения:

- Входной блок (англ. entering block) - базовый блок, не принадлежащий циклу и имеющий ребро, ведущее в заголовок цикла.
- Предзаголовок цикла (англ. preheader) - единственный входной блок цикла.
- Выходящее ребро (англ. exiting edge) - ребро, направленное из блока внутри цикла в блок вне цикла.
- Блок выхода (англ. exit block) - блок, в который ведет выходящее ребро.

1.2 Упрощенное представление циклов

Для упрощения описания трансформаций циклов, в компиляторной инфраструктуре применяются различные внутренние представления. Основное такое представление это упрощенное представление цикла (англ. Loop Simplify Form). Оно обладает следующими свойствами:

- У цикла есть предзаголовок.
- У цикла единственное обратное ребро.
- Все ребра, входящие в блоки выхода, выходят из цикла. Таким образом все блоки выхода доминируются заголовком цикла.

Остальные представления строятся на основе упрощенного представления циклов.

1.3 Расположение инварианта цикла в упрощенном представлении

Рассмотрим варианты расположения инварианта цикла в упрощенном представлении:

- В теле цикла.

  Это расположение зачастую получается при построении внутреннего представления и редко является оптимальным, так как часто число итераций цикла больше единицы.

- В предзаголовке цикла.

  Такое расположение будет оптимальным для цикла, в теле которого используется инвариант, и число итераций которого больше единицы.

- В некоторых блоках выхода цикла.

  Этот вариант расположения доступен только в том случае, если инвариант не используется внутри цикла. Данный подход является оптимальным, при условии минимизации суммарной частоты выходных блоков в которые располагаются копии инварианта и дальнейшей пропагации инструкции без учета циклов.

Универсальным подходом является вынос инварианта из тела цикла в предзаголовок. Однако, как было показано на примере функции `S_regmatch`, этот метод может быть неэффективным если среднее число итераций цикла мало.

1.4 Формальная постановка задачи расположения инвариантов цикла

Рассмотрим задачу оптимального расположения инвариантов цикла формально.

Для каждого цикла, рассматривается множество блоков, доминируемых предзаголовком p

$$D = \{ d : p \, dom \, d \}$$

.

Для каждого инварианта i необходимо найти, такое множество M что:

- $M \subset D$
- $\forall \, u \in U(i) \, \exists \, m \in M : m \, dom \, u$, где $U(i)$ - множество блоков, содержащих использование инварианта i.
- $\sum_{m \in M} f(m) \to min$, где $f(m)$ - оценка частоты вероятности исполнения блока m.

Глава 2

Обзор существующих решений

2.1 Подход к размещению инвариантов цикла в компиляторной инфраструктуре LLVM

Расположение инвариантов цикла в компиляторной инфраструктуре LLVM определяется двумя преобразованиями: вынос инвариантов из цикла и пропагация инвариантов в цикл.

2.1.1 Вынос инвариантов из цикла

Задача данной трансформации - вынести из цикла как можно больше инструкций для облегчения дальнейших оптимизаций циклов.

Дерево циклов функции обходится в порядке снизу - вверх, т.е. подциклы обрабатываются перед циклами. Над каждым из циклов последовательно применяются две трансформации, описанные ниже.

Вынос инструкций в блоки выхода

Сначала все инструкции, не используемые в цикле выносятся в блоки выхода. Это осуществляется следующим образом: Базовые блоки обходятся в порядке, обратном к обходу в ширину дерева доминаторов внутри цикла. Если блок - часть внутреннего цикла - он пропускается, так как он уже был обработан прежде. Для каждой инструкции внутри базового блока:

- Если у инструкции нет использований - она удаляется.
- Если все использования инструкции находятся вне цикла - она может быть перенесена в подмножество выходных блоков, которое

доминирует блоки, содержащие использования результата исполнения инструкции. Такая трансформация может быть применена даже к инструкциям, операнды которых не являются инвариантами цикла, так как на момент использования будет наблюдаться только значение, полученое в последней итерации цикла. Для перемещения инструкции необходимо, чтобы она не имела никаких сторонних эффектов.

Вынос инвариантов цикла в предзаголовок

Остальные инструкции внутри цикла обрабатываются в соответствии со следующим алгоритмом:

- Блоки цикла обходятся в глубину по дереву доминаторов. Блоки находящиеся во внутреннем цикле гнезда пропускаются, так как уже были обработаны прежде.

- Если не обладает сторонними эффектами и все ее операнды являются инвариантами цикла - она тоже является инвариантом цикла и выносится в предзаголовок. Порядок обхода упрощает определение, является ли некоторое значение инвариантом. Так как при описанном выше обходе, к моменту обработки блока, все доминирующие его блоки уже обработаны, инвариант может быть определен как значение, порождающая которое инструкция находится вне цикла.

2.1.2 Пропагация инвариантов в цикл

Эта трансформация применяется уже после осуществления всех цикловых оптимизаций. Фактически, эта трансформация является обратной к выносу инвариантов из цикла в предзаголовок, с учетом того, что пропагируются только те инварианты, для которых можно вследствие этого уменьшить оценку частоты исполнения. Это осуществляется следующим образом: Дерево циклов обходится в том же порядке, что и при выносе инвариантов из цикла. Такой порядок обеспечивает, что все инварианты внутренних циклов гнезда уже расположены оптимально, к моменту расположения инвариантов внешнего цикла. Для каждого цикла строится множество базовых блоков принадлежащих циклу с меньшей частотой, чем у предзаголовка - множество «холодных» блоков. Затем обходятся инструкции предзаголовка. Любая инструкция предзаголовка, не обладающая сторонними эффектами является инвариантом цикла. Для каждой инструкции, являющийся инвариантом, строится множество базовых блоков, содержащих использование этого значения. Суммарная частота базовых блоков, входящих в это множество минимизируется следующим

образом: Для каждого базового блока из множества «холодных» блоков строится пересечение множества блоков, доминируемых «холодным» блоком, с множеством блоков содержащих использование инварианта. Если суммарная частота блоков пересечения больше, чем частота «холодного» блока, эти блоки заменяются на «холодный».

Если суммарная частота полученного множества меньше, чем частота предзаголовка, инвариант копируется во все базовые блоки множества и удаляется из предзаголовка.

Основным ограничением эффективности этой трансформации является точность анализа, предоставляющего частоту каждого блока.

Построение анализа частот исполнения блоков

Частота блоков вычисляется следующим образом:

- Обход всех сильно связных компонент графа потока управления Использует LoopInfo анализ для предоставления информации о компонентах. Компоненты обходятся в таком порядке, что все вложенные компоненты обрападываются перед внешней. Нумерация компонент сохраняется и используется в дальнейшем.

- Распределение весов в циклах

  В случае если сильно связная компонента является гнездом циклов, подциклы обходятся перед основным циклом. Это позволяет представить внктренние циклы как вершины с уже рассчитаным весом. Масса распределяется внутри цикла без учета обратных дуг. Масса заголовок цикла $w_0 = 1$. Для каждой следующей вершины, вес рассчитывается как сумма произведений вероятности перехода в данную вершину из некоторой другой вершины на вес этой вершины.

  $$w_j = \sum_{E(V_i,V_j)} w_i p_{ij}$$

  Порядок обхода обеспечивает, что вес вершины из которой направлена дуга уже рассчитан к моменту рассчета веса вершины в которую направлена дуга. Сумма w_j формируется из дуг внутри подграфа, направленных из уже обработанных блоков в еще не обработанные $(j > i)$. Помимо этого, формируются сумма w_b, отвечающая дугам направленым обратно в предзаголовок. Если существует дуга, направленная из уже обработанного блока в уже обработанный блок $(j < i)$, это свидетельствует о наличии несократимого потока управления – сильно связной компоненты с многими точками входа. В этом случае алгоритм перезапускается, представляя каждую точку входа как заголовок цикла, при этом вся сильно связная компонента представляется набором таких циклов.

- Рассчет числа итераций циклов

  Число итераций цикла рассчитывается как:

$$s = \frac{1}{w_0 - w_b}$$

- Распределение веса в функции

  После того, как все циклы функции скомпанованы, вес внутри функции распределяется аналогично алгоритму распределения веса внутри циклов, учитывая что в полученном представлении функции нет обратных и выходящих дуг.

- Рассчет частот исполнения блоков

  Все циклы функции обходятся в обратном порядке, так что внешний цикл обрабатывается перед внутренним. Для блоков функции частота совпадает с весом. Для каждого блока внутри некоторого цикла, частота получается умножением числа итераций на частоту блока, который представлял скомпанованый цикл.

Для осуществления анализа частот исполнения блоков необходимо обладать информацией о вероятности перехода по каждой из дуг графа потока исполнения. Эта информация предоставляется анализом вероятности переходов.

Построение анализа вероятности переходов

Цель анализа вероятности переходов – построить соответствие между ребром графа потока исполнения и вероятностью перейти по этому ребру в ходе исполнения программы.

В ходе построения анализа все базовые блоки внутри функции обходятся в обратном порядке. Если для базового блока существуют метаданные, предоставляющие информацию о числе переходов по каждой из дуг, собранную в ходе динамического профилирования или полученную каким-либо образом из фронтенда, построение вероятностей перехода тривиально:

$$p_{ij} = c_j / \sum_k c_{ik}$$

Где c_k - число переходов по дуге из i в k.

Если таких метаданных для блока нет, или если данный блок не был посещен ни разу в ходе исполнения программы для сборки профиля, последовательно применяются эвристики, пока некоторая из них не будет применима к данному блоку.

Первой применяется эвристика основанная на весах блоков Предварительно, для всех специфичных базовых блоков функции, таких как блоки, в которых есть функция помеченная как не возвращающая управление или «холодная», или блоки помеченные как недостижимые, или блоки, участвующие в обработке исключений (терминатор непосредственного доминатора - инструкция invoke), присваивается некоторый специфичный малый вес. Всем дугам, выходящим из блока присваевается вес по умолчанию. Для блоков внутри цикла, в отношении которых можно доказать, что условие становиться константой после некоторой итерации, вес дуги считается вдвое меньшим, чем для остальных блоков. Для дуг, которые выходят из цикла, вес считается меньшим в оценку числа итераций раз. Если существует выходящая дуга, вес которой отличен от веса по умолчанию по результату применения приведенных выше эвристик для определения весов, вероятность всех выходных дуг определяется как отношение веса дуги к суммарному весу.

Затем для базовых блоков терминатор которых является инструкцией сравнения применяются следующий ряд эвристик:

- Сравнение указателей

  Применяется для случая сравнения на равенство двух указателей, или сравнения указателя с нулем.

- Сравнение с нулем

  Применяется для сравнения знаковой целой переменной с константой, при условии, что переменная не является результатом применения битовой маски. Если при этом переменная является результатом вызова одной из известных функций стандартной библиотеки такой, как strcmp, рассматриваются только сравнения на равенство или неравенство. Равенство числа нулю считается менее вероятным, чем неравенство. Считается менее вероятным, что число меньше или меньше либо равно нулю.

- Сравнение чисел с плавающей точкой

  Равенство двух чисел с плавающей точкой, каждое из которых не является нечислом (англ. NaN, not a number), считается менее вероятным, чем неравенство. То, что число с плавающей точкой является нечислом, считается маловероятным, так как часто служит для обработки исключительных ситуаций.

Значения вероятностей для этих эвристик представлены в таблице 2.1.

Эвристика	**Условие**	**Вероятность истинности условия**
Сравнение указателей	`p1 == p2`	0.375
	`p1 != p2`	0.625
Сравнение с константой	`i == 0`	0.375
	`i <0`	
	`i == -1`	
	`i <= 0`	
	`i != 0`	0.625
	`i >0`	
	`i != -1`	
	`i >= 0`	
Сравнение чисел с плавающей точкой	`f1 == f2`	0.375
	`f1 != f2`	0.625
	`!isnan(f)`	$1 - 2^{-20}$
	`isnan(f)`	2^{-20}

Таблица 2.1: Эвристики сравнений

Динамическое профилирование

Как видно из описания алгоритма построения анализа вероятности переходов, приведенного выше, единственным способом получить точные значения вероятности переходов, а следовательно и частоты исполнения базовых блоков является использование динамического профиля исполнения программы.

Динамический профиль состоит из значений счетчиков числа переходов по каждой дуге графа потока исполнения. Для получения данных значений код программы должен быть предварительно инструментирован и запущен. Точность профиля напрямую зависит от того, насколько совпадают входные данные программы при запуске, который создает профиль, и запуске на котором производятся измерения.

Для использования динамического профиля необходимо, чтобы внутреннее представление программы совпадало в момент инструментации кода и в момент использования профиля для аннотации дуг графа потока исполнения соответствующим этой дуге числом переходов.

Инструментация кода

Инструментация кода для сбора динамического профиля основана на использовании минимального остовного дерева, описанного в работе Дональда Э. Кнута, и Фрэнсиса Р. Стивенсона [4]. Этот алгоритм минимизирует число инструментируемых ребер графа, используя тот факт, что для каждой вершины, за исключением входной и выходной, сум-

ма счетчиков, на ребрах входящих в вершину, равна сумме счетчиков на выходящих из вершины ребрах. Таким образом, значения счетчиков на остовном дереве, могут быть неявно получены из значений счетчиков на ребрах не входящих в остовное дерево. В работе доказывается, что инструментирование, построенное таким образом, инструментирует минимально возможное число ребер.

Ребра остовного дерева выбираются так, что их частота максимальна. Для этого используется статический анализ частоты переходов. Это позволяет минимизировать значения счетчиков, а следовательно и влияние на производительность программы при сборе профиля.

2.2 Сравнение с подходом к размещению инвариантов цикла в компиляторной инфраструктуре GCC

Подход к размещению инвариантов цикла в компиляторной инфраструктуре GCC во многом аналогичен. Однако, можно выделить несколько принципиальных отличий:

- Вынос инвариантов цикла не является приведением к канонической форме. Это позволяет не разделять алгоритм расположения инвариантов цикла на вынос и пропагацию, а обеспечивать эффективное расположение инвариантов в рамках единственной трансформации. Таким образом, алгоритм может работать быстрее, так как каждый инвариант обрабатывается строго один раз, в то время как в компиляторной инфраструктуре LLVM некоторые инварианты сначала выносятся в предзаголовок, а затем пропагируются обратно. В то же время, наличие инвариантов в теле цикла усложняет и замедляет работу многочисленных цикловых оптимизаций. Поэтому нельзя однозначно оценить, какой подход лучше.

- Статический анализ частот обеспечивает более точную оценку частоты исполнения базовых блоков. Это достигается за счет:

  – Использование большего количества эвристик. Мотивация для использования именно этих эвристик и их значения описаны в работе [5].

  – Использование алгоритма объединения эвристик описанного в работах [6] и [7].

Различия в точности анализа частот исполнения базовых блоков частично объясняют разрыв в эффективности компиляторов Clang и GCC

на примере функции `S_regmatch` из главы 1. Несмотря на это, как будет показано в дальнейшем, инварианты располагаются не оптимально в компиляторной инфраструктуре LLVM и при наличии динамического профиля, в случае чего различия в построении статического анализа не важны.

Глава 3

Исследование и построение решения задачи

3.1 Анализ функции `S_regmatch`

Анализ приведенной в главе 1 функции `S_regmatch` из бенчмарка `perlbench_r` который является частью набора бенчмарка SPEC CPU® 2017, показал:

- Вследствие внутренней структуры функции, во вложенных циклах, образованных безусловными переходами внутри некоторых блоков `case` конструкции `switch` используется большое количество значений, являющихся инвариантами для внутреннего цикла, но не являющихся инвариантами для внешнего цикла.

- Все инструкции порождающие такие значения, выносятся в заголовок внешнего цикла в ходе выноса инвариантов из циклов, так как он является предзаголовком для внутренних циклов и эти значения используются внутри тела внутреннего цикла, а следовательно не могут быть пропагированы в блоки выхода.

- Анализ частоты исполнения базовых блоков показывает, что частота исполнения заголовка внешнего цикла много больше частоты исполнения блоков внутренних циклов, и для многих инструкций, упомянутых выше, эффективна пропагация в тело внутреннего цикла. Это следует из контекста применения данной функции. `S_regmatch` является функцией проверки на совпадение строки и скомпилированного регулярного выражения. Все обсуждаемые выше внутренние циклы являются обработкой особых токенов, таких как токен предпросмотра. Таким образом, можно проследить,

что такие токены редки в скомпилированном выражении, т.е. p_i - вероятность перейти из конструкции `switch` в блок `case i`, отвечающий внутреннему циклу, и число итераций внутреннего цикла c_i - величина порядка единицы. Число вариантов в конструкции `switch`, напротив, велико, а частота исполнения блока цикла не больше частоты исполнения заголовка цикла, при условии отсутствия внутренних циклов, которую можно оценить как:

$$f_i = f_o p_i^{c_i} c_i$$

Где f_o - частота заголовка внешнего цикла, в котором до пропагации инвариантов в тело цикла находятся обсуждаемые инструкции. Таким образом, условие при котором, пропагация в заголовок внутреннего цикла будет эффективна:

$$p_i^{c_i} c_i \leq 1$$

Из структуры неравенства видно, что оно будет часто выполняться, при принятых условиях на p_i и c_i. При наличии в теле внутреннего цикла ветвлений и условии использования инварианта только на некоторых путях, аналогичное выражение принимает вид:

$$p_i^{c_i} c_i \sum_j w_j \leq 1, \sum_j w_j < 1$$

Где w_j - отношение частоты исполнения блока j к частоте исполнения заголовка внутреннего цикла.

- В ходе пропагации инвариантов в тело цикла, многие такие инструкции не меняют своего расположения, несмотря на то, что для этих значений пропагация будет выгодна с точки зрения минимизации суммарной частоты исполнения блоков расположения инварианта.

Из сказанного выше следует, что разрыв в производительности с компилятором GCC, обусловлен субоптимальным алгоритмом пропагации инвариантов в тело цикла. Поэтому задача улучшения алгоритма расположения инвариантов цикла сводится к улучшению именно алгоритма пропагации инвариантов в тело цикла.

3.1.1 Упрощенный пример `S_regmatch_draft`

Для демонстрации рассмотрим более простую функцию, обладающую схожими свойствами. Листинг кода этой функции представлен в приложении 4.1. Упрощенная визуализация графа потока управления функции `S_regmatch_draft` представлена на рисунке 3.1.

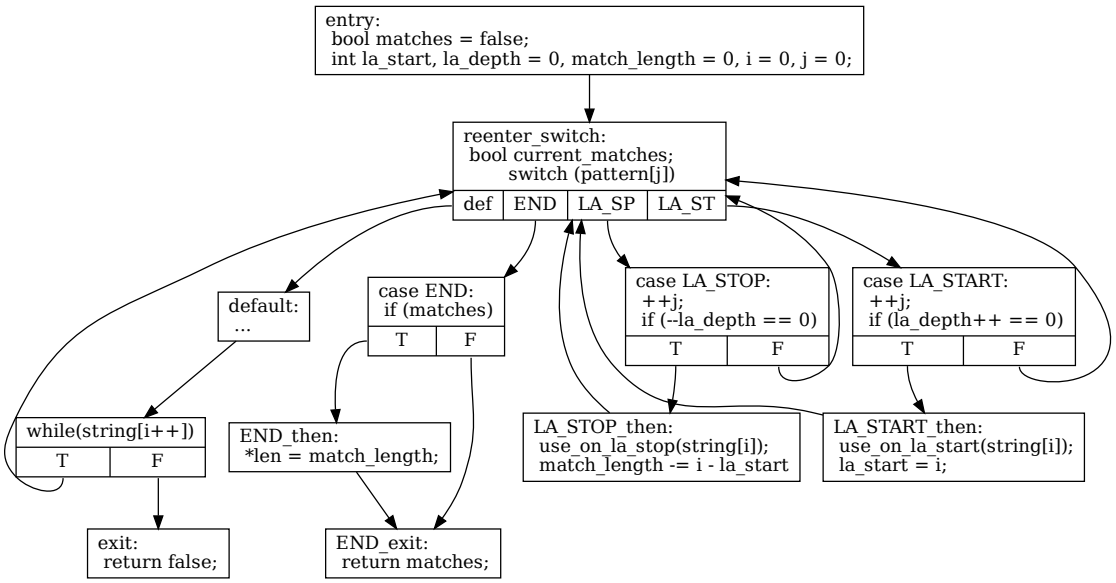

Рис. 3.1: Схема потока упровления в функции `S_regmatch_draft`

Эта функция предназначена для нахождения первого вхождения шаблона `pattern` в строке `string`. Возвращаемое значение – флаг, найден ли шаблон или нет. Позиция начала подстроки, совпадающей с шаблоном и её длина возвращаются через указатели `pos` и `len` соответственно. В шаблоне, помимо символов, могут встречаться специальные токены `LA_STOP` и `LA_START`. Часть шаблона, заключенная между этими токенами - окно предпросмотра. Подстрока совпадает с шаблоном, оканчивающимся окном предросмотра, только если подстрока совпадает с частью шаблона, предшествующей окну предпросмотра, и продолжение подстроки совпадает с окном предпросмотра.

Здесь можно выделить:

- Внешний цикл:

  `reenter_switch` → `default` → `while` → `reenter_switch`

- Внутренние циклы:

  1. `reenter_switch` → `case LA_START` → `LA_START_then` →
     → `reenter_switch`
  2. `reenter_switch` → `case LA_START` → `reenter_switch`
  3. `reenter_switch` → `case LA_STOP` → `LA_STOP_then` →
     → `reenter_switch`
  4. `reenter_switch` → `case LA_STOP` → `reenter_switch`

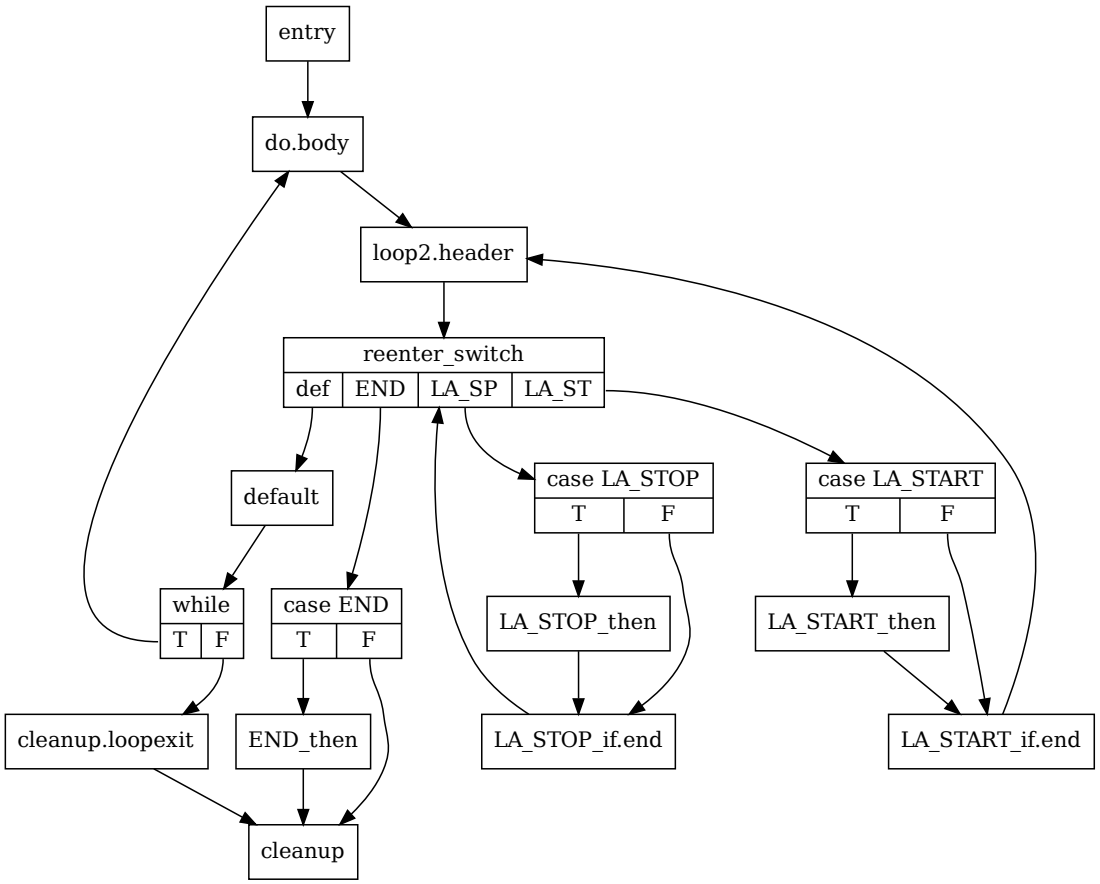

Рис. 3.2: Построение упрощенной формы циклов в функции `S_regmatch_draft`

При более детальном рассотрении этих циклов, можно заметить, что инварианты этих циклов обладают схожими свойствами с инвариантами внутренних циклов функции `S_regmatch`. Так, например, символ `string[i]`, как и сама переменная `i`, индуктивная для внешнего цикла, являются инвариантами для внутренних циклов.

Рассмотрим построение упрощенной формы циклов (англ. Loop Simplify) для этого графа. Схема графа потока управления функции после приведения внутреннего цикла 2 к упрощенной форме представлена на рисунке 3.2. В ходе преобразования был выделен предзаголовок цикла 2 `do.body` и заголовок `loop2.header`.

После завершения построения упрощенной формы циклов для всех циклов функции, граф потока управления будет иметь вид, представленый на рисунке 3.3.

Из рисунка видно, что после построения упрощенной формы циклов, каждый из вложенных циклов обладает:

- Заголовком (`loop1-3.header` и `reenter_switch` для циклов 1-4 соответственно).

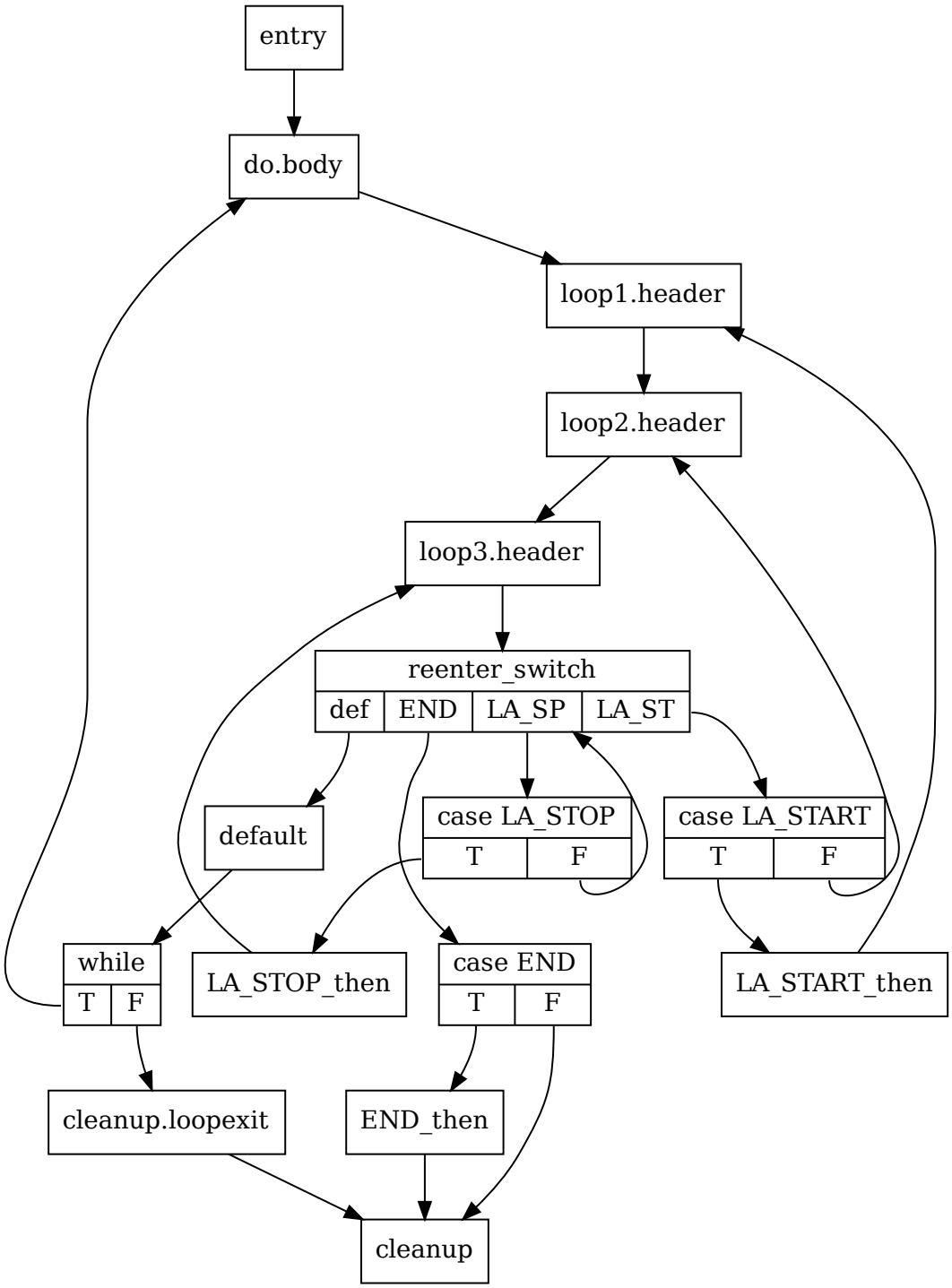

Рис. 3.3: Упрощенная форма циклов функции `S_regmatch_draft`

- Предзаголовком (`do.body` и `loop1-3.header` для циклов 1-4 соответственно).

Таким образом форма была успешно построена и в функции нет несократимого потока управления.

Рассмотрим теперь расположение инварианта `string[i]`. После выноса инвариантов из цикла, он будет расположен в предзаголовке цикла 1 `do.body`. Если этот инвариант не будет пропагирован в тела внутренних циклов, число исполнений интрукций пораждающих этот инвариант за один вход в функцию совпадет с числом итераций внешнего цикла, а именно с длинной строки `string`. Если же он будет пропагирован в базовые блоки `LA_START_then` и `LA_STOP_then`, анализ исходного кода может показать, что соответствующее число исполнений составит сумму числа токенов `LA_START` и `LA_STOP` в шаблоне `pattern`. Из контекста применения функции очевидно, что длина шаблонна, а следовательно и сумма числа токенов `LA_START` и `LA_STOP` в шаблоне, меньше, чем длина строки, в которой ищется совпадающая с шаблоном подстрока. Таким образом, расположение инварианта в блоках `LA_START_then` и `LA_STOP_then` будет оптимальным с точки зрения минимизации числа исполняемых инструкций.

3.2 Улучшения алгоритма пропогации инвариантов в тело цикла

При детальном рассмотрении используемого алгоритма пропагации были замечены две возможности для улучшения:

3.2.1 Пропагация инвариантов с использованием в φ узлах

Многие инварианты цикла могут быть использованы в некотором φ узле. Пример такого цикла приведен на рисунке 3.4. В таком случае, невозможно переместить инвариант в базовый блок в котором находится этот φ узел. Следует рассматривать как кандидата для пропагации не базовый блок содержащий сам узел, а базовый блок из которого поток управления входит в узел с данным значением. Это допустимо, так как использование значения инварианта происходит не в самом базовом блоке, в котором находится φ узел, а на ведущем в него ребре. Соответственно, если определение значения происходит в базовом блоке, из которого выходит ребро, значение инварианта определено к моменту входа в φ узел и цепь определение-использование не нарушена.

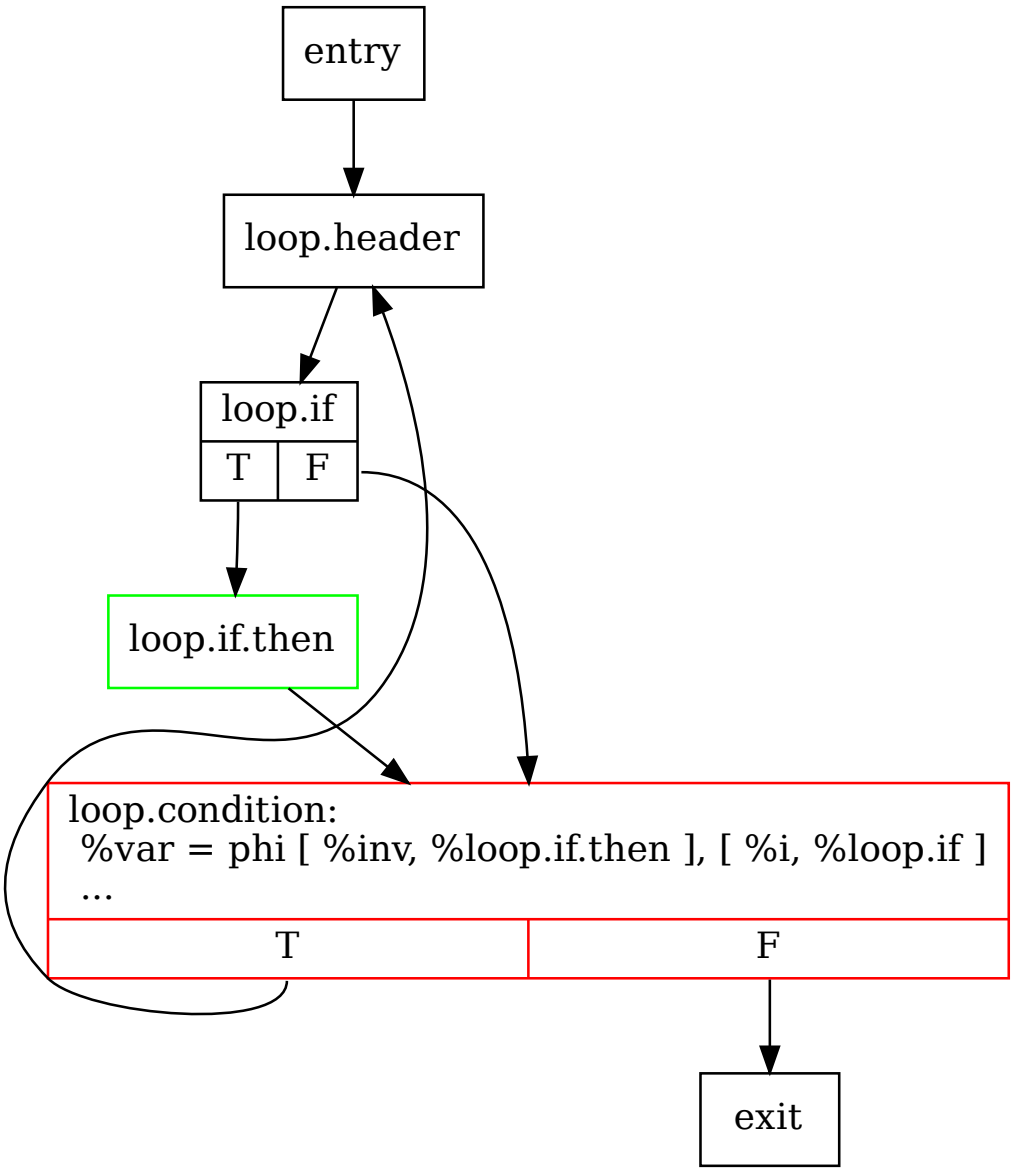

Рис. 3.4: Пример графа потока управления цикла с использованием инварианта в φ-узле

3.2.2 Пропагация инвариантов во все доминируемые базовые блоки

Некоторые использования инварианта могут находиться вне цикла. В таком случае можно рассматривать в качестве кандидатов для пропагации объединение множества базовых блоков цикла с базовыми блоками выхода из цикла. Это допустимо, так как объединение множеств базовых блоков, доминируемых блоками выхода из цикла, совпадает с множеством базовых блоков, доминируемых предзаголовком, за вычетом множества блоков цикла. Это следует из того, что любой путь из предзаголовка в базовый блок вне цикла, пройдет через некоторый блок выхода. Такой подход не изменит ассимптотику алгоритма, так как число выходных блоков, можно ограничить сверху как произведение максимального числа выходящих ребер из блока цикла на число блоков цикла. При этом, число выходящих ребер - константа, определяемая терминатором базового блока.

Такой подход к решению проблемы пропагации инвариантов цикла с использованиями в базовых блоках не принадлежащих циклу является корректным, но не обеспечивает расположения инвариантов, оптимального с точки зрения минимизации оценки суммарной частоты исполнения.

Это можно показать на контрпримере, граф потока управления которого представлен на рисунке 3.5 в упрощенной форме циклов. Рассмотрим функцию с циклом, в котором существует ветвление потока управления и лишь одна из ветвей содержит использование инварианта. Блоки циклов доминируют над подграфом, в котором существует аналогичное ветвление.

Рассмотрим частоты исполнения блоков функции.

Ввдем следующие обозначения:

- p_{if} – вероятность перехода из блока `if` в блок `if.then`.
- $p_{loop.if}$ – вероятность перехода из блока `loop.if` в блок `loop.if.then`.
- c – число итераций цикла.
- f_{entry} – частота исполнения предзаголовка цикла `entry`
- $f_{loop.if.then}$ – частота исполнения блока цикла `loop.if.then`
- f_{if} – частота исполнения блока выхода цикла `if`
- $f_{if.then}$ – частота исполнения базового блока `if.then`

Частота предзаголовка цикла принимается равной единице.

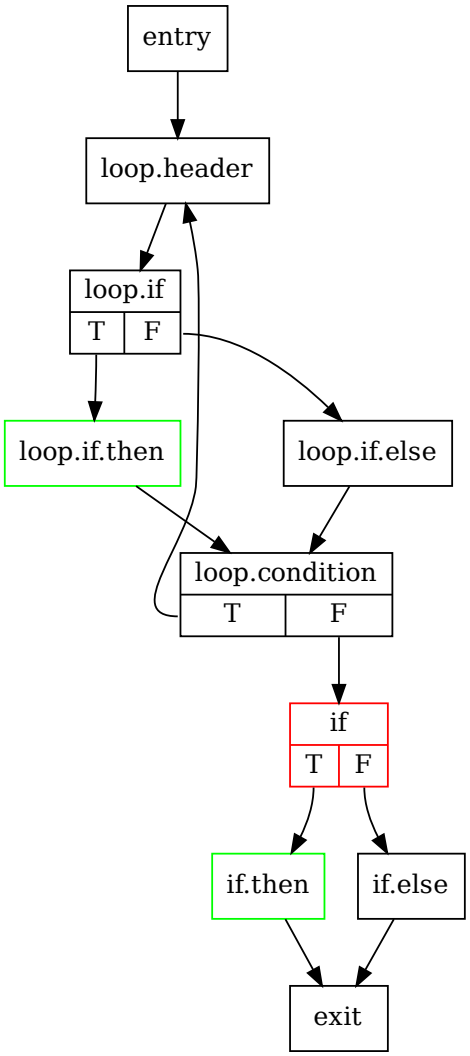

Рис. 3.5: Контрпример графа потока управления, к пропагации в объединение множеств блоков цикла и блоков выхода

$$f_{entry} = 1$$

Частота базового блока `if` равна частоте предзаголовка, так как этот базовый блок является единственным блоком выхода цикла.

$$f_{if} = f_{entry} = 1$$

Остальные частоты, введенные выше, выражаются следующим образом:

$$f_{if.then} = f_{if} p_{if} = p_{if}$$

$$f_{loop.if.then} = f_{entry} c\, p_{loop.if} = c\, p_{loop.if}$$

Пусть существует инвариант цикла, используемый в базовых блоках `loop.if.then` и `if.then`. Этот инвариант будет находиться в предзаголовке цикла `entry` после выноса инвариантов из тела цикла, так как он используется в блоке цикла `loop.if.then`.

Суммарная частота иполнения блоков `loop.if.then` и `if.then` составляет:

$$f_1 = f_{if.then} + f_{loop.if.then} = p_{if} + c\, p_{loop.if}$$

Пусть выполняется следующее условие на вероятности переходов:

$$p_e + c\, p_l < 1$$

В таком случае, суммарная частота исполнения базовых блоков в которых расположены инструкции, порождающие инвариант, может быть минимизирована при пропагации инварианта в базовые блоки `loop.if.then` и `if.then`.

Однако, если ограничиться при объединением множеств блоков цикла с блоками выхода цикла для рассмотрения как кандидатов в целевые блоки пропагации, пропагация не будет эффективна, так как суммарная частота исполнения блока выхода цикла и любого другого блока будет больше, чем частота исполнения предзаголовка:

$$f_2 = f_{if} + f = 1 + f > 1 = f_{entry}$$

Таким образом, при условии использования инварианта вне цикла, для минимизации суммарной частоты исполнения базовых блоков расположения инварианта, не достаточно ограничиваться рассмотрением как кандидатов в целевые блоки пропагации объединения множеств блоков цикла с блоками выхода цикла, а следует рассматривать все базовые блоки, доминируемые предзаголовком.

3.3 Оценка эффективности предлагаемого алгоритма расположения инвариантов цикла

Рассмотрим расположение инвариантов цикла в компиляторной инфраструктуре LLVM после применения описанного выше алгоритма, модифицированного предложенным образом, а именно:

- В ходе выноса инвариантов из цикла инвариант был:
  - Удален, если у него нет использований.
  - Вынесен в подмножество блоков выхода, если все использования инварианта находятся вне цикла.
  - Вынесен в предзаголовок цикла, если есть блоки цикла использующие значение инварианта.
- В ходе пропагации инвариантов в тело цикла:
  - Для цикла было построено множество C, состоящее из базовых блоков, доминируемых предзаголовком и имеющих меньшую оценку частоты исполнения.
  - Для каждого инварианта цикла было построено множество U, состоящее из базовых блоков, содержащих его использование.
  - Для каждого цикла строится множество:

$$M : M \subset C,\ \forall\, u \in U\ \exists\, m \in M : m\ dom\ u$$

  - Оценка суммарной частота базовых блоков множества M минимизруется.
  - Инвариант был спропагирован в базовые блоки множества M, если оно существует, и итоговая оценка суммарной частоты исполнения оказалась меньше, чем оценка частоты исполнения предзаголовка.

Таким образом, для инвариантов цикла существует 4 варианта расположения, которые характеризуются в обозначениях формальной постановки задачи из главы 1 следующим образом:

1. Инвариант удален:
$$M = U = \emptyset$$
2. Инвариант расположен в подмножестве блоков выхода.

3. Инвариант расположен в предзаголовке цикла p.

4. Инвариант расположен в множестве M, полученом в ходе пропогации инвариантов в тело цикла.

Проверим эти варианты расположения на соответствие условиям, изложенным в постановке задачи в главе 1.

- $M \subset D = \{\, d : p \; dom \; d \,\}$
  - Вариант 1 – выполняется, так как:
    $$M = \emptyset \in D$$
  - Вариант 2 – выполняется, так как предзаголовок доминирует все блоки выхода.
  - Вариант 3 – выполняется, так как:
    $$p \; dom \; p \; \forall \; p$$
  - Вариант 4 – выполняется, так как:
    $$M \subset C \subset D$$
- $\forall \, u \in U(i) \; \exists \, m \in M : m \; dom \; u$
  - Вариант 1 – выполняется, так как:
    $$U = \emptyset$$
  - Вариант 2 – выполняется по построению.
  - Вариант 3 – выполняется, так как:
    $$p \; dom \; b$$
    Где b - базовый блок, в котором инвариант был расположен изначально.
  - Вариант 4 – выполняется по построению.
- $\sum_{m \in M} f(m) \rightarrow min$
  - Вариант 1 – выполняется, так как сумма равна нулю.
  - Вариант 2 – не выполняется, так как можно построить контр-пример: Пусть терминатор блока выхода является условным переходом, и инвариант используется только в одном из непосредственно доминируемых базовых блоков. Тогда, размещение инварианта непосредственно в блоке с использованием будет обладать меньшей частотой исполнения.

– Вариант 3 – выполняется, по построению варианта 4:

– Вариант 4 – выполняется по построению.

Таким образом, доказано, что полученное расположение корректно и, за исключением случая выноса в выходные блоки, оптимально. Однако, инструкция, вынесенная в выходной блок не имеет использований в цикле, и может быть пропагирована в последствии в рамках трансформации, которая будет решать задачу минимизации частоты исполнения инструкции на ациклическом графе, представляя каждый цикл его предзаголовком, из которого поток управления переходит в блоки выхода. Эта трансформация реализована в компиляторной инфраструктуре LLVM.

Приведенное выше рассуждение доказывает, что предложенный алгоритм обеспечивает оптимальное расположение инвариантов цикла.

Глава 4

Описание практической части

4.1 Реализация предложенных улучшений

В этом разделе описана реализация алгоритма пропагации инвариантов цикла в тело цикла с учетом предложенных улучшений в компиляторной инфраструктуре LLVM.

4.1.1 Обработка циклов функции

Циклы внутри функции обходятся в следующем порядке: все вложенные циклы обрабатываются перед обработкой внешнего цикла.

Для каждого цикла строится множество «холодных» блоков – базовых блоков, доминируемых предзаголовком и имеющих оценку частоты исполнения меньше, чем у предзаголовка. Это множество представлено в коде программы как две структуры: сортированный по возрастанию частоты базового блока массив указателей на блоки и хеш-таблицы нумерации «холодных» блоков, которая предоставляет соответствие между значением указателя и индексом в массиве. Хеш-таблица используется впоследствии для обеспечения определенности порядка исполнения программы.

После чего алгоритм переходит к обработке инструкций предзаголовка цикла.

4.1.2 Обработка инструкций предзаголовка цикла

Инструкции предзаголовка обходятся в обратном порядке. Это упрощает сохранение зависимостей между инвариантами. Пусть существуют два инварианта a и b. b использует значение a, а следовательно, b расположен ближе к началу предзаголовка, чем a. Тогда если инвариант a бу-

дет спропагирован раньше, чем b, цепь определение-использование будет нарушена в ходе трансформации, и обеспечения корректности потребует дополнительных операций. Для каждой инструкции, не имеющей сторонних эффектов:

- Строится множество базовых блоков, содержащих использование инструкции, или, в случае если инструкция используется в φ-узле, блоков, из которого поток управления приходит в φ-узел с значением инварианта.
- Затем строится множество базовых блоков, в которые будет производиться пропагация M.
  - M инициализируется как копия множества блоков использований инварианта.
  - Для каждого блока из множества «холодных» базовых блоков:
    * Строится множество блоков из M, которые доминируются «холодным» блоком.
    * Если суммарная частота базовых блоков этого множества больше, чем частота «холодного» блока, блоки заменяются в множестве M на «холодный» блок.
- Если суммарная частота блоков множества M больше частоты предзаголовка, инвариант не пропагируется.
- Если в M единственный блок, инвариант пропагируется в этот блок.
- Иначе, указатели на базовые блоки из множества M копируются в массив. Этот массив сортируется по возрастанию значений хеш-таблицы нумерации «холодных» блоков.
- В каждый блок из полученного массива вставляется копия инструкции, порождающей инвариант, и все использования оригинальной инструкции в доминируемых блоках заменяются на использование копии, оригинальная инструкция удаляется.

4.2 Анализ асимптотики алгоритма

В этом разделе рассчитывается асимптотика описанного алгоритма.

4.2.1 Асимптотика времени построения множества M

Построение множества M имеет асимптотику $O(|U||C|)$, где C – множество «холодных» базовых блоков, U – множество базовых блоков, содержащих использование инварианта. Оба этих числа могут иметь порядок числа блоков цикла. Поэтому, для ограничения времени исполнения трансформации линейной функцией от числа блоков цикла, вводится дополнительный параметр $|U|_{max}$. Если $|U|$ превышает значение этого параметра, такой инвариант не рассматривается. Это не оказывает сильного влияние на эффективность алгоритма, так как для инварианта с большим числом использований маловероятно найти множество M, удовлетворяющее поставленным требованиям.

4.2.2 Асимптотика времени пропагации инварианта

Копирование инструкции в блоки множества M и замена использований инструкции занимает:

$$O(|M|N_u)$$

Где N_u - число использований инструкции.

По построению множества M:

$$|M| \leq |U|$$

При этом $|U|$ ограничен сверху константой $|U|_{max}$.

Строго говоря, число использований может нелинейно зависеть от числа блоков, содержащих использования $|U|$. Однако, его можно ограничить сверху произведением общего числа инструкций в блоках множества U на константу C – максимальным числом аргументов инструкции:

$$N_u \leq C \sum_{b \in U} N_b$$

Тогда выражение 4.2.2 приводится к виду:

$$O(|U|_{max}\, C \sum_{b \in U} N_b) = O(\sum_{b \in U} N_b)$$

4.2.3 Асимптотика времени работы над циклом

Из рассуждений выше, следует, что асимптотика времени исполнения алгоритма над некоторым циклом будет равна:

$$O(N_p \sum_{i \in P} \sum_{b \in U_i} N_b)$$

Где:

- N_p – число инструкций предзаголовка.
- P – множество инструкций предзаголовка.
- U_i – множество блоков, содержащих использование инварианта i.

Аналогично 4.2.2:

$$\sum_{i \in P} \sum_{b \in U_i} N_b \leq C\ N$$

Где N – число инструкций в функции.

Тогда 4.2.3 приводится к виду:

$$O(N_p\ N)$$

4.2.4 Асимптотика времени обработки функции

Асимптотика времени обработки функции имеет вид:

$$O(N \sum_{l \in L} N_{p_l})$$

Где L – множество циклов функции, N_{p_l} – число инструкций в предзаголовке цикла l.

Каждый цикл обладает уникальным предзаголовком, а следовательно:

$$\sum_{l \in L} N_{p_l} \leq N$$

Таким образом, в худшем случае, асимптотика алгоритма имеет вид:

$$O(N^2)$$

В среднем, число циклов и инструкций в предзаголовке цикла много меньше числа общего числа инструкций функции.

$$|L| << N, N_{p_l} << N$$

Исходя из этого утверждения, средняя асимптотика времени работы алгоритма:

$$O(N)$$

4.3 Анализ производительности

Для анализа производительности произведенных изменений были измерены изменения числа исполненых инструкций на наборе бенчмарков SPEC CPU® 2017 и наборе бенчмарков из коллекции тестов LLVM.

4.3.1 Методика измерений

Измерения проводились на системе на кристале Alibaba T-Head XuanTie C910 ICE под управлением операционной системы Линукс. Процессор этой системы имеет 2 ядра. Все бенчмарки запускались в однопоточном режиме, второе ядро системы не нагружалось. Тактовая частота процессора была зафиксирована и составляла 1.2 ГГц. Процессор имеет архитектуру набора команд RISC-V 64 GC.

Измерялись время исполнения программы в тактах и число исполненных инструкций. Сравнение времени исполнения программы позволяет оценить влияние улучшения на общую производительность целевой платформы. Сравнение числа исполненных инструкций позволяет выделить влияние платформонезависимой оптимизации, так как не учитывает среднее время исполнения инструкции, которое в большей мере зависит от микроархитектуры процессора, чем от исполняемого кода.

Для сбора метрик использовалась программа perf. Эта программа позволяет собирать значения соответствующих счетчиков процессора, тем самым обеспечивая наиболее точные значения собираемых метрик при непосредственном исполнении на целевой машине.

Анализ производительности трансформации проводился с использованием динамического профилирования. Как было показано в главе 2, это позволяет получить наиболее точную информацию о частоте исполнения базовых блоков, и тем самым, оценить производительность алгоритма расположения инвариантов отдельно от точности анализа вероятности переходов.

Измерения производились следующим образом:

- Код программы компилировался с включенной инструментацией кода для генерации динамического профиля.
- Полученный исполняемый файл запускался для сбора профиля исполнения программы.
- Код программы компилировался повторно, с использованием собранного профиля.
- Полученный исполняемый файл запускался под управлением программы perf для сбора метрик.

4.3.2 SPEC CPU® 2017

Результат вышеописанных измерений для набора бенчмарков SPEC CPU® 2017 представлен на графиках 4.1.

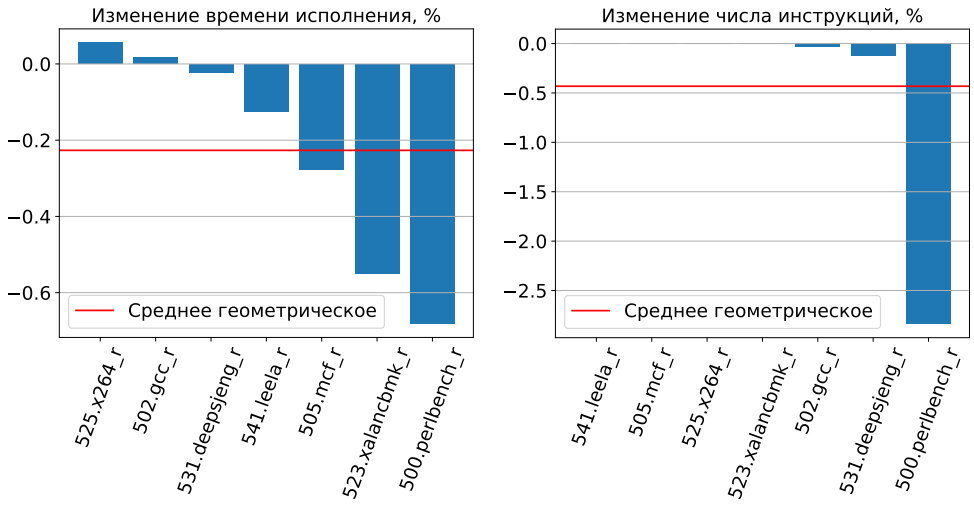

Рис. 4.1: Сравнение производительности на SPEC CPU® 2017

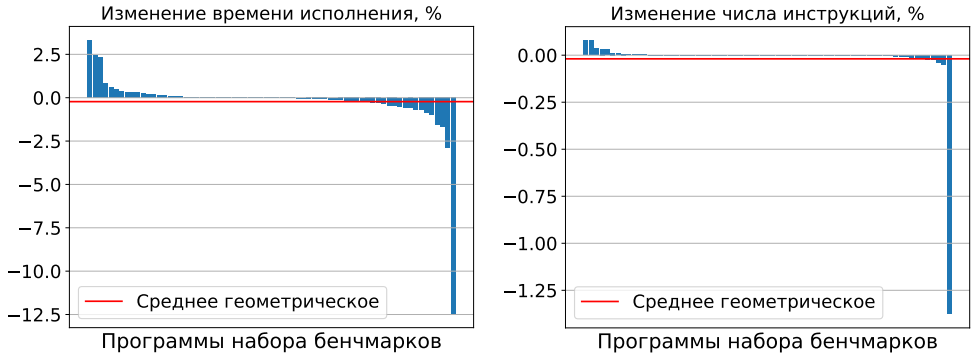

Рис. 4.2: Сравнение производительности на наборе из коллекции тестов LLVM

На графике видно уменьшение по обеим метрикам, особенно на бенчмарке «500.perlbench_r», что свидетельствует об эффективности произведенных улучшений.

4.3.3 Коллекция тестов LLVM

Бенчмарки из этого набора имеют значительно меньший размер, поэтому, для уменьшения погрешности, вносимой операционной системой, собиралось среднее значение метрик по результатам нескольких последовательных запусков набора бенчмарков. Результат измерений представлен на графиках 4.2. На графиках программы набора отсортированы по убыванию значения метрики.

Можно видеть, что в большинстве случаев, метрики убывают. Рост числа инструкций на некоторых тестах связан с неточностью профиля и погрешностью измерений. Среднее геометрическое изменений значений метрики меньше нуля для каждой метрики, что свидетельствует об улучшении производительности.

Заключение

Как отмечено во введении, алгоритм расположения инвариантов цикла в компиляторной инфраструктуре LLVM был недостаточно эффективен. Была поставлена задача улучшения этого алгоритма.

Задача решена в полном объеме, а именно:

- В ходе работы работе был проведен общий анализ алгоритма расположения инвариантов цикла, описанный в главе 2, и анализ его применения к функции `S_regmatch`, на которой наблюдался значительный разрыв в производительности (глава 3). В результате этого анализа, было показано, что причиной неэффективности являются недостатки в алгоритме пропагации инвариантов в тело цикла.

- Были разработаны улучшения алгоритма пропагации инвариантов в тело цикла – пропагация инвариантов с использованием в φ узлах и пропагация инвариантов во все доминируемые базовые блоки, описанные в главе 3.

- Для улучшенного алгоритма, была доказана оптимальность получаемого расположения инвариантов.

- Алгоритм был реализован в компиляторной инфраструктуре LLVM. Асимптотика времени исполнения алгоритма является в среднем линейной, а в худшем случае квадратичной, от числа инструкций функции, как показано в главе 4.

- Проведен анализ производительности алгоритма на наборе бенчмарков SPEC CPU® 2017 и наборе бенчмарков из коллекции тестов LLVM. Этот анализ показал, что предложенные улучшения алгоритма:

  – Значительно увеличивают производительность некоторых приложений – До 12.5% и 2.5% по времени исполнения и числу исполненных инструкций соответственно.

  – Обеспечивают средний прирост производительности на используемом наборе бенчмарков.

Предлагаемые изменения алгоритма пропагации инвариантов цикла были включены во внутреннюю поставку компилятора компании Синтакор.

Литература

[1] *Aho, Alfred Vaino.* The Running Time of Programs / Alfred Vaino Aho, Jeffrey David Ullman // Foundations of Computer Science. — TPB, 2004.

[2] Перемещение кода / Альфред Ахо, Моника С. Лам, Рави Сети, Джеффри Ульман // Компиляторы: принципы, технологии, инструменты. — Вильямс, 2008. — P. 714–715.

[3] *Muchnick, Steven S.* Loops and Strongly Connected Components / Steven S. Muchnick // Advanced compiler design and implementation. — Morgan Kaufmann, 1997. — P. 191–196.

[4] *Knuth, Donald E.* Optimal measurement points for program frequency counts / Donald E. Knuth, Francis R. Stevenson // *BIT*. — 1973. — Vol. 13, no. 3. — P. 313–322.

[5] *Ball, Thomas.* Branch prediction for free / Thomas Ball, James R. Larus // *ACM SIGPLAN Notices*. — 1993. — Vol. 28, no. 6. — P. 300–313.

[6] *Wu, Youfeng.* Static branch frequency and program profile analysis / Youfeng Wu, J.R. Larus // *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture.*

[7] Corpus-based Static Branch prediction / Brad Calder, Dirk Grunwald, Donald Lindsay et al. // *ACM SIGPLAN Notices*. — 1995. — Vol. 30, no. 6. — P. 79–92.

[8] *Lattner, C.* LLVM: a compilation framework for lifelong program analysis & transformation / C. Lattner, V. Adve // International Symposium on Code Generation and Optimization, 2004. CGO 2004. — 2004. — Pp. 75–86.

[9] *Lattner, Chris Arthur.* LLVM: An infrastructure for multi-stage optimization: Ph.D. thesis / University of Illinois at Urbana-Champaign. — 2002.

[10] *Hennessy, John L.* Measuring, Reporting, and Summarizing Performance / John L. Hennessy, David A. Patterson, Krste Asanovic // Computer Architecture: A quantitative approach. — Morgan Kaufmann, 2011. — P. 36–44.

Приложения

```
#include "S_regmatch_draft.h"
#include <stdio.h>

#ifdef DEBUG
#define LOG_DEBUG(fmt, Args...) printf(fmt "\n", Args);
#else
#define LOG_DEBUG(Args...) ;
#endif

bool S_regmatch_draft(const char *string, const enum Token *pattern,
                      int *pos, int *len) {
  bool matches = false;
  int la_start, la_depth = 0, match_length = 0, i = 0, j = 0;
  do {
    LOG_DEBUG("string[%d] == %c", i, string[i]);
  reenter_switch:
    LOG_DEBUG("matching token %s", token_name(pattern[j]));
    bool current_matches;
    switch (pattern[j]) {
    case END:
      if (matches) {
        LOG_DEBUG("match of length %d detected at [%d, %d)",
                  match_length, *pos, i);
        *len = match_length;
      }
      return matches;
    case LA_START:
      ++j;
      if (la_depth++ == 0) {
        use_on_la_start(string[i]);
        la_start = i;
      }
      goto reenter_switch;
    case LA_STOP:
      ++j;
      if (--la_depth == 0) {
        use_on_la_stop(string[i]);
        match_length -= i - la_start;
      }
      goto reenter_switch;
    default:
      current_matches = pattern[j] == string[i];
      if (!matches && current_matches) {
        LOG_DEBUG("match started at %d", i);
        *pos = i;
      } else if (matches && !current_matches) {
        LOG_DEBUG("mismatch at %d", i);
        match_length = 0;
        la_depth = 0;
        j = 0;
      }
      matches = current_matches;
      if (matches) {
        ++j;
        ++match_length;
      }
    }
  } while (string[i++]);
```

```
  return false;
}
```

Листинг 4.1: Реализация функции S_regmatch_darft на языке программирования C

```
#pragma once
#include <limits.h>
#include <stdbool.h>

enum Token {
  LA_START = CHAR_MAX,
  LA_STOP,
  END,
};

const char *token_name(enum Token token);

void use_on_la_start(char c);

void use_on_la_stop(char c);

bool S_regmatch_draft(const char *string, const enum Token *pattern,
                      int *pos, int *len);
```

Листинг 4.2: Заголовочный файл `S_regmatch_draft.h`