НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. Н.И. ЛОБАЧЕВСКОГО
ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ
ЛАБОРАТОРИЯ «ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»

ПРОЕКТ «ИССЛЕДОВАТЕЛЬСКИЙ КОМПИЛЯТОР»

Практикум
«Оптимизирующие компиляторы»
(на примере GCC)

Содержание

Предисловие

Пособие предназначено студентам, желающим самостоятельно изучить основополагающие технологии современных оптимизирующих компиляторов. В центре внимания находится промышленный компилятор GNU Compiler Collection (GCC). Пособие основано на материалах семинаров «Проблемы генерации кода в компиляторе», проводимых в учебно-исследовательской лаборатории "Информационные технологии" факультета ВМК (проект «Исследовательский компилятор»). Целью цикла семинаров и данного документа является создание у студента связного представления об архитектуре современного промышленного компилятора (на примере GNU GCC). Цикл семинаров расширен серией компьютерных лабораторных работ, посвященных практическому изучению компилятора GCC (добавление собственного языка, перенацеливание на новую архитектуру, добавление прохода оптимизации). Авторы предполагают, что читатель знаком с теорией формальных языков (на ВМК эта теория излагается в курсах «Теория автоматов и мат. логика» Д.И. Когана и «Сетевые грамматики и языковые процессоры» С.Г. Кузина), а также с классическими принципами построения компиляторов, см., например, [1]. Для выполнения лабораторных работ требуется, чтобы у слушателя был опыт программирования на языке Си и некоторая практика пользовательской работы в среде операционной системы Linux.

Разработка методического пособия выполнена в рамках проекта «Исследовательский компилятор».

В заключение предисловия авторы выражают благодарность В.П. Гергелю, Н.Ю. Золотых, Л.В. Нестеренко за неоценимую помощь в разработке данного пособия.

Общая структура компилятора

Наиболее общее определение для понятия компилятор таково:

Компилятор – это программа, которая получает на входе программу, написанную на одном языке – *исходном*, и транслирует (переводит) её в эквивалентную программу на другом языке – *целевом*.

Рисунок 1. Общая структура простого (не оптимизирующего) компилятора

Для нас интерес представляет более узкое определение этого термина:

Компилятор – это программная система, которая переводит программы, написанные на языках высокого уровня в объектный или машинный код для выполнения на вычислительной машине.

Опишем общую структуру компилятора (рисунок 1):

- *Лексический анализатор.* Преобразует поток символов, который представляет исходный текст программы в поток токенов. Т. е. программа разбивается на «слова» исходного языка, называемые лексемами.
- *Синтаксический анализатор*. Получает на вход последовательность токенов и, обычно основываясь на контекстно-свободных грамматиках, генерирует некоторое промежуточное представление, например в виде дерева. В процессе этого преобразования формируется таблица символов.
- *Семантический анализатор.* Проверяет семантику программы, т.е. определяет правильность написания программы, основываясь на правилах исходного языка, которые не могут быть выражены контекстно-свободными грамматиками. Важным аспектом этого этапа является проверка типов.
- *Кодогенератор.* Преобразует программу на промежуточном языке в перемещаемый машинный код или ассемблерный код. Для каждой переменной программы определяется её положение в памяти. Каждая промежуточная инструкция транслируется в одну или несколько машинных инструкций. Ключевой аспект этой фазы заключается в назначении переменных регистрам.

В настоящее время представляют интерес только, так называемые, оптимизирующие компиляторы. *Оптимизирующим компилятором* называется такой компилятор, который применяет улучшающие в некотором смысле преобразования программы, именуемые *оптимизациями*. Цели оптимизации могут быть различными, например, уменьшение времени исполнения программы и/или уменьшение размера кода.

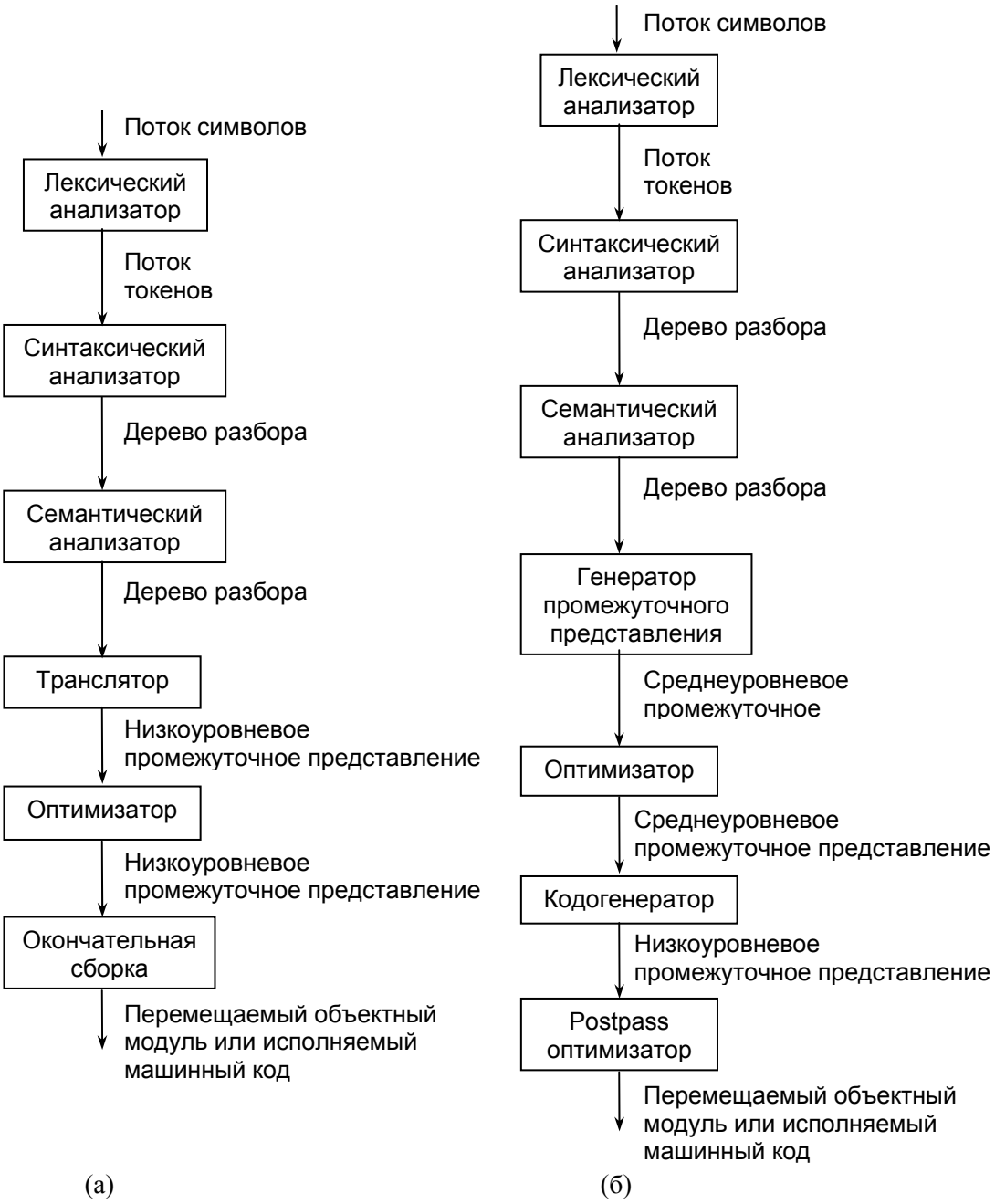


Рисунок 2. Две структуры оптимизирующего компилятора:
а) низкоуровневая модель,
б) смешанная модель

На рисунке 2 приведены две основные схемы оптимизирующих компиляторов. В первом случае (рисунок 2.а) все оптимизирующие преобразования выполняются над промежуточным представлением низкого уровня. Во втором (рисунок 2.б) – исходная программа транслируется в среднеуровневое представление, над которым происходит большинство архитектурно независимых оптимизаций, а затем,

переводится в низкоуровневое представление, над которым проводят машинно-зависимые оптимизации.

Как видно, работа компилятора разделяется на несколько фаз. Но, при реализации часто происходит объединение действий, выполняемых в различных фазах. Так, например, фазы объединяются в *начальную стадию*, или *frond end*, и *заключительную стадию*, или *back end*. Начальная стадия состоит из тех этапов компиляции, которые зависят от исходного языка и почти не зависят от целевой платформы (лексический и синтаксический анализ, создание таблицы символов, семантический анализ и генерация

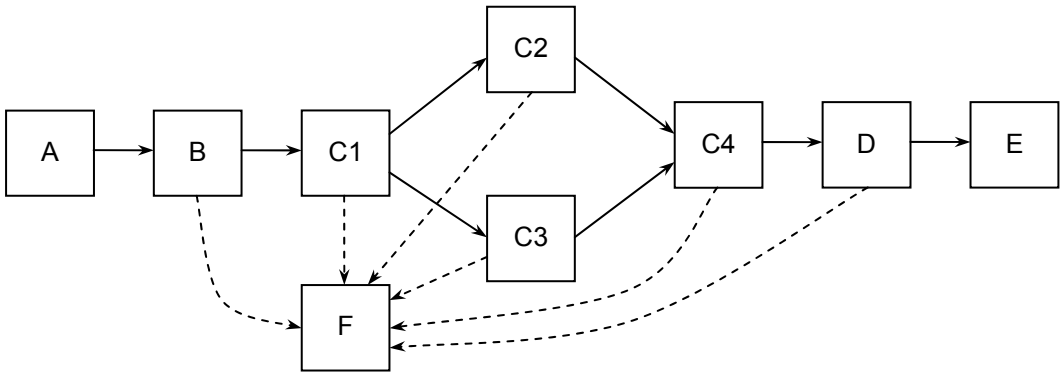


Рисунок 3. Порядок оптимизаций в компиляторе

промежуточного кода). Сюда так же может относиться некоторая часть оптимизатора.

Заключительная стадия, состоит из тех этапов, которые зависят от целевой архитектуры, для которой выполняется компиляция. В эту стадию входит часть оптимизации кода и генерация выходного кода, сопровождаемые необходимой обработкой ошибок и работой с таблицей символов.

Теперь рассмотрим подробнее структуру оптимизатора. Представим оптимизации в виде групп, и покажем, в каком порядке следует применять эти группы (Рисунок 3). Раскроем состав и назначение каждой из групп.

A. Эти оптимизации обычно производятся над каким-либо промежуточным представлением высокого уровня. Это делается для того, что бы сохранить исходную структуру циклов, последовательность операций в них и доступ к элементам массивов в виде близком к исходному. В эту группу входят следующие оптимизации:

- замена скалярными переменными ссылок на элементы массива;
- оптимизации кэша данных.

B, C. Оптимизации данной группы производятся над промежуточным представлением среднего или низкого уровня (в зависимости от выбранной модели оптимизаций: низко-уровневой или смешанной – см. рисунок 2). Разветвление от группы C1 к группам C2 и C3 представляет выбор метода оптимизаций, которые состоят в том, что бы уменьшить частоту выполнения некоторых частей кода без изменения семантики программы. Это разветвление так же представляет выбор анализа потока управления для применения оптимизаций. Перечислим оптимизации, входящие в рассматриваемые блоки:

- B
    - встраивание функций;
    - оптимизации хвостовых вызовов, включая удаление хвостовой рекурсии;
    - замена агрегатов на константу;
    - продвижение констант в условных операторах;
    - межпроцедурное продвижение констант;
    - специализация и клонирование процедур;
- C1
    - глобальное номерование значений;

    - глобальное и локальное распространение копий;
    - продвижение констант в условных операторах;
    - удаление «мёртвого кода»;
- C2
    - локальное и глобальное удаление общих подвыражений;
    - вынесение инвариантного кода из тела цикла;
- C3
    - частичное удаление ненужных вычислений;
- C4
    - удалёние «мёртвого кода»;
    - подстановка операторов;
    - снижение мощности выражений с индексной переменной;
    - замена линейных индексов на адресные выражения;
    - удаление индексной переменной;
    - устранение лишних проверок включения в диапазон значений;
    - оптимизации потока управления;

D. Эти оптимизации всегда проводятся над низкоуровневой формой промежуточного представления, так как они могут быть полностью машинно-зависимыми. В эту группу входят следующие оптимизации:

- низкоуровневое встраивание функций;
- оптимизация функций, не содержащих вызовов;
- оптимизация кода пролога и эпилога функций, содержащих вызовы других функций;
- специфические машинные оптимизации;

- совмещение концов базовых блоков;
- оптимизация ветвлений и перемещение условных выражений;
- удаление «мёртвого кода»;
- программная конвейеризация с раскруткой циклов, variable expansion, переименование регистров, hierarchical reduction;
- планирование инструкций – 1;
- распределение регистров с помощью раскраски графа;
- планирование инструкций – 2;
- внутрепроцедурная оптимизация кэша инструкций;
- упреждающая выборка инструкций;
- упреждающая выборка данных;
- предсказание переходов.

E. Оптимизации этой группы производятся на этапе линковки кода; они оперируют перемещаемым объектным кодом:

- межпроцедурное распределение регистров;
- агрегация глобальных объектов;
- межпроцедурная оптимизация кэша инструкций.

F. Данный блок содержит следующие оптимизации:

- свёртывание константных выражений;
- алгебраические упрощения.

Эта группа связана с другими блоками на рисунке 2 пунктирными линиями, что означает возможность применение этих оптимизаций на любом другом этапе – там, где это понадобится.

GNU Compiler Collection

Для проведения исследований в области компиляции или в целях обучения принципам работы современного промышленного компилятора, имеет смысл обратиться к GNU Compiler Collection. У GCC имеется несколько достоинств, которые позволяют легко использовать его в этих целях.

Разработка собственного компилятора требует больших затрат времени и ресурсов, а также знаний, которых может не быть у разработчиков. Покупка коммерческого компилятора приведёт к значительным материальным затратам, что не всегда приемлемо, а в случае обучения даже не всегда возможно. В большинстве случаев, наиболее рациональным выходом для исследователей и разработчиков оказывается использование и модификация открытого компилятора.

Несомненно, в настоящее время, среди свободно распространяемых открытых компиляторов самым развитым является компилятор GCC. Это перенацеливаемый (как по входному языку, так и по целевой архитектуре) компилятор доступный по лицензии GPL.

Оригинальная версия компилятора была написана Ричардом Сталлманом (Richard Stallman). Сейчас, существует огромное сообщество разработчиков, которые используют и совершенствуют GCC.

Отметим некоторые характерные черты данного компилятора:

- Поддерживает большое число языков и машинных архитектур:
    - Языки: C, C++, Ada95 (GNAT), Fortran 77, Fortran 95, Pascal, Modula-2, Modula-3, Java, Cobol, Chill (Cygnus).

  - Архитектуры: ARM, Alpha (DEC), Intel x86, Motorola 680x0, 68C11, DSP56000, Hitachi SH и H8300, MIPS, IBM PowerPC, HP PA-RISC, SUN SPARC, IA64, AMD x86-64.

- По качеству не уступает многим известным компиляторам (в том числе и коммерческим). Так, на Alpha результаты работы GCC сравнимы с компилятором от DEC.
- GCC является постоянно развивающимся проектом. Отметим основные направления работы по развитию компилятора:
  - реализация новых языков программирования;
  - реализация новых алгоритмов оптимизации;
  - введение поддержки новых платформ;
  - улучшение библиотек времени исполнения;
  - ускорение процесса отладки.

Проходы GCC

Компиляция в GCC производится *проходами*, то есть последовательностью преобразований исходного кода программы во внутреннее представление, оптимизацией внутреннего представления, и преобразования внутреннего представления в машинный код.

Всего проходов в GCC 3.4 – 26 штук, начиная от лексико-синтаксического разбора и заканчивая генерацией машинного кода и отладочной информации.

Рассмотрим подробнее последовательность проходов. Они производятся друг за другом, но выполнение части из них может быть пропущено, если необходимость в проходах отсутствует. Управление выполнением проходов производится драйвером *gcc*, который описан в файле `gcc.h`.

Отдельно для каждого прохода описаны улучшения кода, производимые при проходе, файлы, содержащие исходный код и ключи компиляции, влияющие на исполнение прохода.

Парсер (лексический и синтаксический анализатор)

В этом проходе происходит чтение содержимого описания входной программы, и создается представление функций в виде дерева. Также в этом проходе производятся семантический анализ и языкозависимый анализ типов данных, при этом каждому узлу дерева, представляющему выражение, присваивается тип данных. Переменные представляются как узлы дерева-декларации. В результате прохода получается представление программы в виде древовидной структуры. Оптимизации на этом этапе не происходит.

- *Файлы, описывающие проход: tree.c, fold-const.c, stor-layout.c (языконезависимый разбор); tree.h, tree.def (описание формата древовидного представления программы); c-* (синтаксический разбор С-программ).*

Оптимизация дерева

Оптимизация представления программы в виде дерева, перед его переводом во внутреннее представление в виде RTL. В настоящий момент основная оптимизация, выполняющаяся на этом этапе – встраивание вызываемых функций (inlining).

Эта оптимизация позволяет увеличить скорость выполнения программы за счет уменьшения числа вызовов функций.

- *Реализация функциональности находится в файле tree-inline.c.*

Так же выполняется свертывание констант (поддерево – выражение над константами – преобразуется в один узел-константу) и упрощение некоторых арифметических выражений.

Оптимизация позволяет увеличить скорость выполнения программы за счет отсутствия вычисления константных выражений и многократного использования констант.

- *Реализация методов находится в файле fold-const.c.*

Генерация RTL

Преобразование представления программы в виде дерева в RTL код. При этом проходе выполняется оптимизация if-условий для сравнений, булевых операций и условных выражений. Определяется хвостовая рекурсия. Принимается решение о лучшей организации циклов и операторов выбора (switch).

Возможные улучшения работы механизма предсказания переходов.

- *Генерация RTL описана в файлах: stmt.c, calls.c, expr.c, explow.c, expmed.c, function.c, optabs.c, emit-rtl.c.*

В конце генерации RTL принимается решение о возможности встраивания функции. Функция должна удовлетворять некоторым условиям и ограничениям, таким как максимальный размер, число и типы параметров. Функция при этом может содержать циклы и рекурсивные вызовы самой себя.

- *Код для сохранения RTL-кода функции и последующего его встраивания содержится в файле integrate.c.*
- *Ключ для получения отладочной информации прохода:* ***-dr****, файл с отладочной информацией: .rtl.*

Оптимизация вызовов

Происходит удаление рекурсивных вызовов в конце функции и оптимизация хвостовых (sibling) вызовов функций (замена команды вызова на команду перехода). Цель оптимизации – уменьшить накладные расходы при вызове функций (там, где это возможно). Для этого удаляются лишние команды организации кадра стека.

Уменьшение накладных расходов при вызове функций.

- *Реализация располагается в файле sibcall.c.*
- *Ключ для получения отладочной информации:* ***-di****, файл с отладочной информацией: .sibling.*

Оптимизация переходов

Упрощает переходы к следующим инструкциям, цепочки переходов и т. д. Происходит удаление меток, на которые никто не ссылается и удаление недостижимого кода, за исключением случая, когда недостижимый код содержит циклические конструкции. Также происходит преобразование кода, реализованного с переходами, в последовательность инструкций, устанавливающих значения по результатам сравнения, если такие инструкции поддерживаются, например, инструкции set*cc* для Intel® Pentium® II и выше.

Оптимизация переходов производится два или три раза. Первый раз непосредственно после генерации RTL, второй раз после поиска общих подвыражений (если это потребуется), третий раз непосредственно перед генерацией ассемблерного кода.

Возможные улучшения: удаление неиспользуемого кода, упрощение предсказания переходов.

- *Файл с реализацией: jump.c.*
- *Ключ для получения отладочной информации: **-dj**, файл с отладочной информацией: .jump.*

Сканирование регистров (определение времени жизни переменных)

В результате прохода появляется информация о первом и последнем использовании (времени жизни) каждого (псевдо) регистра. Эта информация используется в дальнейшем при удалении общих подвыражений.

- *Файл с реализацией: regclass.c.*

Зацепление переходов

При этом проходе происходит поиск условных переходов, где после проверки условия переход происходит к такому же условию либо к его инверсии. Такие переходы могут быть «сцеплены» по второму условию (так как первую проверку можно опустить). Проход выполняется, если опция `-fthread-jumps` включена.

Возможные улучшения: упрощение предсказания переходов.

- *Файл с реализацией: jump.c*

SSA

Проходы, основанные на использовании представления SSA *(static single assignment form)* включаются опцией `-fssa`. В представлении SSA каждая переменная (псевдорегистр) присваивается только один раз. В настоящий момент преобразование в форму SSA присутствует не во всех версиях. Окончательно оно реализовано в версии 4.

- *Файл с реализацией ssa.c*
- *Ключ для получения отладочной информации:* ***-de****, файл с отладочной информацией: .ssa.*

Продвижение констант в условных операторах

Используется для подстановки констант в условных операторах. Включается опцией `-fssa-ccp`. Время исполнения линейно зависит от размера кода.

- *Ключ для получения отладочной информации:* ***-dW****, файл с отладочной информацией: .ssaccp.*

Удаление «мертвого» кода

Осуществляет удаление неиспользуемого кода, который не оказывает видимого эффекта на программу. Включается опцией `-fssa-dce`. Время исполнения линейно зависит от размера кода.

- *Ключ для получения отладочной информации:* ***-dX****, файл с отладочной информацией: .ssadce.*

Удаление общих подвыражений (CSE)

Проход производит распространение констант и удаление повторно вычисляемых выражений. Если распространение констант станет причиной преобразования условного перехода в безусловный или его ликвидацию, то после CSE запускается оптимизация переходов.

Основная идея CSE – проходя через код функции сохранять представления выражений, производящих одинаковые вычисления и заменять эти выражения самым «дешевым» эквивалентным выражением. Очень сложно

отслеживать разные возможности, когда происходит слияние нескольких ветвей исполнения кода. Поэтому в каждой такой точке слияния обычно все, что было известно о выражениях до этой точки, забывается. Каждый базовый блок обрабатывается отдельно.

- *Файлы с реализацией: cse.c, cselib.c.*
- *Ключ для получения отладочной информации:* ***-ds****, файл с отладочной информацией: .cse*

Глобальное удаление общих подвыражений GCSE

В зависимости от того оптимизируется ли программа по размеру или по скорости, этот проход выполняет одну из двух оптимизаций. Если оптимизация происходит по скорости выполнения, производится LCM (lazy code motion – ленивое перемещение кода). LCM основывается на работе Knoop, Ruthing и Steffen. LCM также обеспечивает вынос инвариантов за пределы цикла. Если происходит оптимизация по размеру, то используется удаление частичной избыточности методом Morel-Renvoise. Этот метод не производит перемещение инвариантов за пределы цикла.

- *Файлы с реализацией: gcse.c, lcm.c.*
- *Ключ для получения отладочной информации:* ***-dG****, файл с отладочной информацией: .gcse.*

Оптимизация циклов

При этом проходе неизменяемые в циклах выражения перемещаются за пределы циклов. Так же возможно снижение мощности выражений и раскрутка циклов.

При втором проходе основное внимание уделяется оптимизации на уровне базовых блоков, кроме того, производится раскрутка, вынесение первых и последних итераций за пределы цикла (peeling) и вынесение условных выражений за пределы цикла.

Возможные улучшения: упрощение циклов, возможно улучшение работы системы предсказания переходов.

- *Файлы с реализацией: loop.c (есть документация), loop.h, unroll.c (есть документация), integrate.c, integrate.h, dependence.h, cfgloopanal.c, cfgloopmanip.c, loop-init.c, loop-unswitch.c, loop-unroll.c.*
- *Ключ для получения отладочной информации: **-dL**, файл с отладочной информацией: .loop, .loop2.*

Оптимизация цепочек переходов

Этот проход – более агрессивная форма глобального CSE. Происходит преобразование управляющего графа функции с помощью распространения констант в условия условных операторов.

- *Файл с реализацией: gcse.c.*
- *Ключ для получения отладочной информации: **-dG**, файл с отладочной информацией: .bypass.*

Вторичное удаление общих подвыражений

Включается опцией `-frerun-cse-after-loop`. Повторный запуск CSE.

- *Ключ для получения отладочной информации: **-dt**, файл с отладочной информацией: .cse2.*

Анализ потока данных

Этот проход делит программу на базовые блоки. В процессе деления удаляются недостижимые циклы. Затем вычисляется время жизни каждого псевдорегистра. Также происходит обнаружение выражений, значения которых далее нигде не используются. Объединяются ссылки на ячейки памяти с инструкциями сложения или вычитания для дальнейшего их преобразования в автоинкрементную или автодекрементную адресацию.

Возможные улучшения: сокращение размера базовых блоков.

- *Файл с реализацией: flow.c.*
- *Ключ для получения отладочной информации: **-df**, файл с отладочной информацией: .flow.*

Комбинирование инструкций

Делается попытка объединить 2 – 3 инструкции, последовательно преобразующие данные, в одну комбинированную инструкцию, если таковая имеется. Для этого объединяются подстановкой RTL выражения для этих инструкций, и производится алгебраическое упрощение выражения. Затем делается попытка сопоставить результат комбинирования инструкций с описанием команды в описании процессора.

Возможные улучшения: использование более сложных инструкций, которые работают быстрее, чем последовательность простых.

- *Файл с реализацией: combine.c.*
- *Ключ для получения отладочной информации: **-dc**, файл с отладочной информацией: .combine.*

Преобразование условных операторов (if-конверсия)

Преобразование зависимостей по управлению в зависимости по данным. Например, преобразование условного кода в поток управления без ветвей – обычно для предикатного исполнения команд, например для архитектуры IA-64.

- *Файл с реализацией: ifcvt.c.*
- *Ключ для получения отладочной информации: **-dE**, файл с отладочной информацией: .ce.*

Перемещение регистров

В этом проходе обрабатываются инструкции, в которых требуется перезагрузить значение из памяти в регистр, но эта перезагрузка может быть произведена с помощью команды «перемещение регистр-регистр». Затем производится попытка оптимизировать код так, чтобы избавится от перемещения регистра – изменив просто номер регистра в инструкции.

Возможные улучшения: уменьшения числа обращений к памяти и перемещения значений из регистра в регистр.

- *Файл с реализацией: regmove.c.*
- *Ключ для получения отладочной информации: **-dN**, файл с отладочной информацией: .regmove.*

Планирование инструкций

Проход определяет инструкции, выходные данные которых в выходных регистрах не будут доступны некоторое количество тактов для последующих инструкций. Делается попытка переупорядочить инструкции в базовом блоке так, чтобы разделить определение и использование выходных данных, в противном случае в конвейере будут приостановки.

Планирование инструкций производится дважды: после комбинирования инструкций и после перезагрузки значений в регистры.

Возможные улучшения: предотвращение задержек в конвейере.

- *Файлы с реализацией: sched-deps.c, sched-ebb.c,sched-int.h, sched-rgn.c, sched-vis.c.*
- *Ключ для получения отладочной информации: **-dS**, файл с отладочной информацией: .sched.*

Распределение регистров

Во время этого прохода удаляются все ссылки на псевдорегистры. Это достигается либо назначением им аппаратного регистра, либо заменой их эквивалентными выражениями (например, константами), либо помещением их в стек. Проход состоит из нескольких подпроходов.

Предпочтение класса регистра

RTL-код сканируется с целью получения информации о том, какой класс регистров (какой регистровый файл) предпочтительнее для каждого псевдорегистра.

- *Файл с реализацией: regclass.c.*

Локальное распределение регистров

Назначает аппаратные регистры псевдорегистрам, используемым внутри одного базового блока.

- *Файл с реализацией: local-alloc.c.*
- *Ключ для получения отладочной информации: **-dl**, файл с отладочной информацией: .lreg.*

Глобальное распределение регистров

Назначает регистры псевдорегистрам, которые используются более чем в одном базовом блоке (внутри одной функции).

- *Файл с реализацией: global.c.*

Распределение регистров с помощью раскраски графа

Другой метод распределения регистров. Включается опцией `-fnew-ra`.

- *Файлы с реализацией: ra.h, ra.c, ra-build.c, ra-colorize.c, ra-debug.c, ra-rewrite.c.*
- *Ключ для получения отладочной информации: **-dl**.*

Перезагрузка регистров

Происходит перенумерование псевдорегистров в соответствии с аппаратными регистрами, которые им назначены. Псевдорегистры, для которых не нашлось аппаратного регистра, помещаются в стек. Далее ведется поиск некорректных инструкций, в которых по разным причинам значение не может быть помещено в регистр или этот регистр имеет несовместимый с типом данных тип. Такие инструкции корректируются загрузкой «проблемных» значений во временные регистры. Для того, чтобы сделать в памяти копию некоторого значения из регистра, генерируются дополнительные инструкции для сброса значений регистров в память.

- *Файлы с реализацией: reload.c, reload.h, reload1.c.*
- *Ключ для получения отладочной информации: **-dg**, файл с отладочной информацией: .greg.*

Планирование инструкций-2

Планирование инструкций повторяется для того, чтобы попытаться исключить приостановки конвейера, вызванные операциями загрузки из памяти, которые появились из-за сброса псевдорегистров в память в предыдущем проходе.

Возможные улучшения: предотвращение задержек в конвейере.

- *Файлы с реализацией: sched-deps.c, sched-ebb.c,sched-int.h, sched-rgn.c, sched-vis.c.*
- *Ключ для получения отладочной информации: **-dR**, файл с отладочной информацией: .sched2.*

Переупорядочивание базовых блоков

Реализуется управляемое профилировщиком переразмещение кода. Если информация профилировщика не доступна, применяются различные методы статического анализа (например, частота запуска базовых блоков или вероятность выбора ветви программы при переходах).

- *Файлы с реализацией: bb-reorder.c, predict.c.*
- *Ключ для получения отладочной информации: **-dB**, файл с отладочной информацией: .bbpro.*

Управление отложенными переходами

Опциональный проход для некоторых процессоров. Пытается найти инструкции, которые могут исполняться в свободных слотах команд переходов, которые допускают отложенную форму (например, сигнальные процессоры, некоторые типы RISC-процессоров).

Возможные улучшения: эффективное использование слотов задержки.

- *Файл с реализацией: reorg.c.*
- *Ключ для получения отладочной информации: **-dd**, файл с отладочной информацией: .dbr.*

Укорачивание ветвей

В некоторых процессорах команды условного перехода имеют ограниченный диапазон перехода (например, от -128 до +127 слов). В случае, если ветвь имеет бо́льшую длину, организовывается более длинная последовательность команд для перехода.

Преобразование регистров в регистровый стек

Преобразование кода базовых блоков – вместо использования обычных регистров с произвольным доступом к использованию регистрового стека. В настоящее время, это поддерживается только для регистров для хранения чисел с плавающей точкой в сопроцессоре Intel 80387.

- *Файл с реализацией: reg-stack.c.*
- *Ключ для получения отладочной информации: **-dk**, файл с отладочной информацией: .stack.*

Вывод ассемблерного кода

Выводит ассемблерный код для функции. Также несёт ответственность за определение ненужных инструкций тестирования и сравнения. Проводится машинно-зависимая оптимизация. Пролог и эпилог функции генерируются в виде ассемблерного кода (они никогда не существуют в RTL-коде).

Возможные улучшения: использование специализированных инструкций.

- *Файл с реализацией: final.c.*

Вывод информации для отладки

- *Файлы с реализацией: dbxout.c, sdbout.c, dwarfout.c, dwarf2out.c, dwarf2asm.c, vmsdbgout.c.*

Резюме по проходам

Таблица 1. Проходы GCC.

Название прохода	Ключи для получения rtl-дампа	Имя файла дампа
Парсер		
Оптимизация дерева		
Генерация RTL	-dr	.rtl
Оптимизация вызовов	-di	.sibling
Оптимизация переходов	-dj	.jump
Сканирование регистров		
Зацепление переходов		
SSA оптимизация	-de	.ssa
Продвижение условных констант	-dW	.ssaccp
Удаление “мертвого” кода	-dX	.ssadce
Удаление общих подвыражений	-ds	.cse
Глобальное удаление общих подвыражений	-dG	.gcse
Оптимизация циклов	-dL	.loop, .loop2
Оптимизация цепочек переходов	-dG	.bypass
Вторичное удаление общих подвыражений	-dt	.cse2
Анализ потока данных	-df	.flow
Комбинирование инструкций	-dc	.combine
Преобразование условных операторов	-dE/-dC	.ce3/.ce1
Перемещение регистров	-dN	.regmove
Планирование инструкций	-dS	.sched
Распределение регистров	-dl	
Предпочтение класса регистра		
Локальное распределение регистров		.lreg
Глобальное распределение регистров		.greg
Распределение регистров с помощью раскраски графа		
Перезагрузка регистров		
Планирование инструкций-2	-dR	.sched2
Переупорядочивание базовых блоков	-dB	.bbpro
Управление отложенными переходами	-dd	.dbr

Укорачивание ветвей		
Преобразование регистров в регистровый стек	-dk	.stack
Вывод ассемблерного кода		
Вывод информации для отладки		

Дополнительные ключи:

Таблица 2. Дополнительные ключи.

Название прохода	Ключи для получения rtl-дампа	Имя файла дампа
Ассемблерный файл аннотируется отладочной информацией	-dA	
Дамп после подсчёта вероятности исполнения ветвей переходов	-db	.bp
Вывод всех макроопределений	-dD	
Вывод после организации кода обработки исключений	-dh	.eh
Вывод после машинно-зависимых оптимизаций	-dM	.mach
Вывод после перенумерования регистров после их распределения регистров	-dn	.rnreg
Вывод после трассировщика (потока данных)	-dT	.tracer
Вывод после второго прохода анализа потока данных	-dw	.flow2
Вывод после прохода pipeline-оптимизации	-dz	.peephole2
Вывод вообще всей отладочной информации обо всех проходах	-da	
Вывод статистики использования памяти	-dm	
Аннотация ассемблерных инструкции с показом возможных вариантов генерации кода	-dp	
Вывод RTL в выходном тексте программы на ассемблере	-dP	
Вывод после каждого прохода информации об управляющем графе в виде, пригодном для просмотра с помощью программы просмотра графов VCG	-dv	
Вывод информации при синтаксическом и семантическом анализе на дескриптор устройства ошибки	-dy	

Представление программ в компиляторе

При преобразовании программы, написанной на языке программирования высокого уровня, в ассемблерный код, выполняемом при компиляции, исходная текстовая форма представления преобразуется в форму, удобную для обработки компилятором. Промежуточная форма увеличивает эффективность работы компилятора, позволяет более точно проводить анализ потока данных и потока управления и реализовывать разнообразные преобразования. Выбор соответствующего промежуточного представления оказывает определяющее влияние на реализацию и сложность исполнения оптимизирующих преобразований, и, в итоге, на время компиляции. В целом, адекватное внутреннее представление программы позволяет улучшить многие характеристики инфраструктуры компилятора, например:

1. продлить жизненный цикл компилятора, в основном ядра, выполняющего различные преобразования внутреннего представления программы;
2. возможность задействовать оптимизатор, максимально (машинно) независимый от формы представления программы и от характеристик компьютера;
3. возможносмть использования компилятора как перенацеливаемого – генерирующего машинный код для разных архитектур.

Обычно к внутреннему представлению программы предъявляется ряд требований, среди ключевых можно выделить:

1. выразимость оптимизации – явное указание в промежуточном представлении программы действий и конструкций исходной программы;
2. сохранение качества – переход к внутреннему представлению не должен нарушать исходные свойства программы, допускающие эффективную реализацию;
3. сохранение свойств (присутствующих во входной программе и полезных при выполнении оптимизаций);
4. унификация конструкций промежуточной программы по отношению к набору применяемых оптимизаций;
5. удобство оптимизации с точки зрения упрощения алгоритмов оптимизатора.

К наиболее распространённым формам промежуточного представления программ относятся синтаксическое дерево; управляющий граф (уграф), который может иметь разные формы – например, плоскую или иерархическую; постфиксная нотация (для шитых кодов); трёхадресный код.

С практической точки зрения хорошее промежуточное представление должно удовлетворять ещё ряду требований:

1. исполняемость – возможность отследить ход исполнения программы;
2. возможность добавлять к структурам данных различную информацию о зависимостях по данным, управлению, статистике исполнения и т.д.;
3. представление циклов в явном виде;

4. компактность (трансляция средней по размеру программы может требовать до 16Мбайт ОЗУ, очень большой – может превысить адресное пространство процесса в 32-х разрядной модели памяти).

К наиболее совершенным на сегодняшний день формам можно отнести управляющий граф программы (содержащий в себе информацию о зависимостях по данным и управлению). В большинстве компиляторов используется граф зависимостей по управлению, дополнительно содержащий в себе графы зависимостей по данным.

RTL

Большая часть работы компилятора GCC осуществляется над промежуточным представлением называемым *Register Transfer Language (RTL)*.

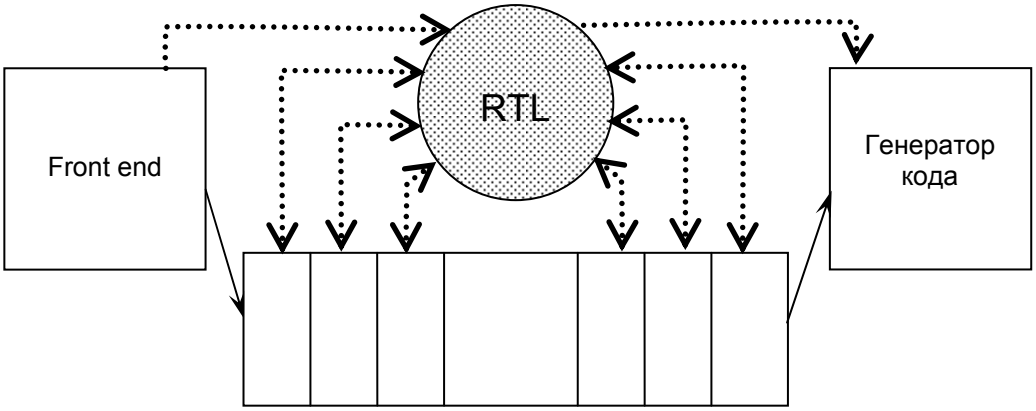


Рисунок 4

Register Transfer Language используется для представления инструкций. При этом инструкции подробно описываются в духе списка LISP.

RTL оперирует пятью типами объектов:

- Выражения (expressions)
- Целые числа (integers)
- Длинное целое (wide integers)
- Строки (strings)
- Вектора (vectors)

RTL формат

Каждый элемент RTL имеет:

- GET_CODE: код операции
  - GET_RTX_CLASS: тип RTL
  - GET_RTX_FORMAT: строка с типом каждого параметра
  - GET_RTX_LENGTH: количество операндов
- GET_MODE: режим
  - SImode: 4-х байтное целое
  - QImode: 1 байтное целое
  - SFmode: 4-х байтное с плавающей точкой
- Список операндов
- Флаги

Пример RTL

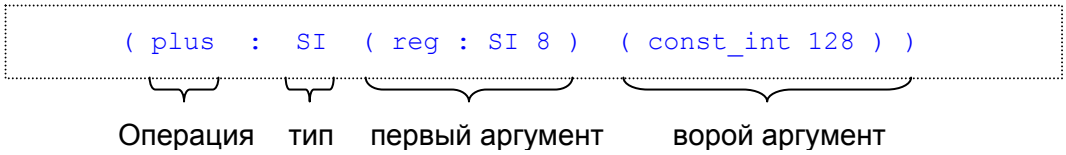


Рисунок 5

- Суммирует два операнда *(plus : SI (<операнд 1>) (<операнд 2>))*
  - Операнды рассматриваются как четырехбайтные целые *(plus : **SI** ...)*
- Первый операнд – это регистр *(**reg** : SI 8)*
  - Регистр хранит 4-х байтное целое *(reg : **SI** 8)*

- Номер регистра – 8 *(reg : SI **8**)*

- Второй операнд – целое число *(**const_int** 128)*
  - Значение – число ‘128’ *(const_int **128**)*
  - Режим VOIDmode (не указан)

Представление констант в RTL

Подробный список допустимых описаний вы можете найти в разделе Constant Expression Types главы RTL Representation в документации GCC Internals.

Мы рассмотрим:

(const_int i)	Этот тип выражения представляет целочисленное значение i. Пример: (const_int 128)
(const_vector:m [x0 x1 …])	Представляет вектор констант. В квадратных скобках указаны значения, хранящиеся в векторе. M собственно указывает тип значений.

В GCC Internals вы найдете описание для: *const_int, const_double, const_vector, const_string, symbol_ref, label_ref, const, high*.

Представление регистров и памяти в RTL

Подробный список допустимых описаний вы можете найти в разделе Registers and Memory главы RTL Representation в документации GCC Internals.

Мы рассмотрим:

(reg:m n)	Для малых значений n (меньших константы FIRST_PSEUDO_REGISTER) эта запись будет означать ссылку на конкретный аппаратный регистр. Для больших значений n это будет означать временное значение или псевдорегистр.
(mem:m addr alias)	Означает ссылку на основную память по адресу addr. M означает тип объекта к которому ведется обращение (размер). Alias означает имя переменной.

В GCC Internals вы найдете описание для: *reg, subreg, scratch, cc0, pc, mem, addressof*.

Представление арифметических выражений в RTL

Подробный список допустимых описаний вы можете найти в разделе RTL Expressions for Arithmetic главы RTL Representation в документации GCC Internals.

Мы рассмотрим:

(plus:m x y)	Представляет сумму значений представленных x и y для типа m.
(compare:m x y)	Представляет результат вычитания y из x с целью сравнения.

В GCC Internals вы найдете описание для: *plus, lo_som, minus, ss_plus, us_plus, ss_minus, us_minus, compare, neg, mult, div, udiv, mod, umod, smin, smax umin, umax not, and, ior, xor, ashift, lshiftrt, ashiftrt, rotate, rotatert, abs, sqrt, ffs, clz, ctz, popcount, parity*.

Прочие инструкции

За информацией по представлению других инструкций обращайтесь непосредственно к GCC Internals: Представление операций сравнения в RTL, Представление битовых полей в RTL, Представление векторных операций, Представление преобразований типов, Представление векторных операций, Представление процедур и функций, Представление векторных операций, Ассемблерные инструкции, Insns, Вызовы функций.

SSA форма в GCC

Static Single Assignment Form (SSA Form) – форма представления программы в которой любой переменной значение присваивается не более одного раза.

SSA форма и граф управления потоком были предложены для представления потока данных и потока управления в программе. Каждая из этих ранее независимых технологий использовалась в классе оптимизаций. Большое число современных алгоритмов оптимизации программ основаны на совместном использовании графа управления и SSA-формы.

Минусы представления RTL и дерева, используемого front end, как промежуточных представлений при работе оптимизатора

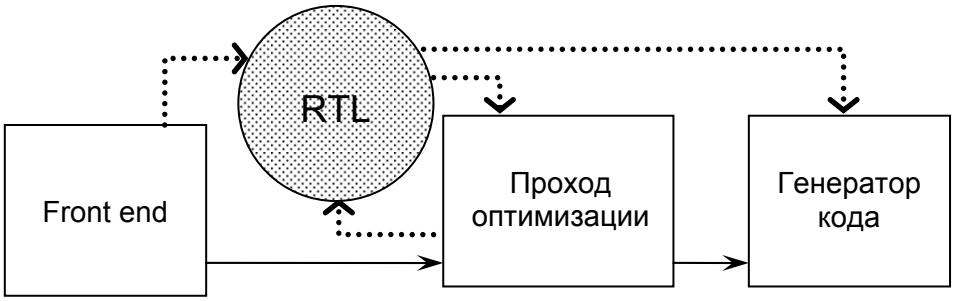


Рисунок 6

Большинство проходов работают с RTL представлением.

Почему ни RTL представление, ни дерево в Front end не подходит для современных методов оптимизации.

RTL обладает рядом минусов. Основные из них:

- Не подходит для высокоуровневых преобразований

- Потеряна оригинальная структура программы

Можно попробовать проводить оптимизацию на деревьях Front end. Ряд алгоритмов так и делают. Особенно, если оптимизация зависит от языка программирования.

Деревья:

- Отражают структуру исходной программы
- Поддерживают высокоуровневые (близкие к исходному коду) преобразования

НО:

- Каждый front end использует свой диалект дерева.

Представление Tree SSA

Tree SSA – это новая система оптимизации основанная на SSA представлении и действующая на GCC Tree представлении. Идея заключается в использовании специального представление в целом очень похожего на дерево, используемое в Front end, но унифицированное для всех поддерживаемых языков. Разработчики обещают внедрить данное представление в ближайших версиях GCC. Мы рассмотрим корни этой идеи: SSA-представление и управляющий граф.

SSA

SSA – это форма программы в которой значение каждой переменной присваивается только один раз, но может читаться сколько угодно раз.

На рисунках 7 и 8 показаны типичные примеры преобразования в SSA форму.

Рисунок 7

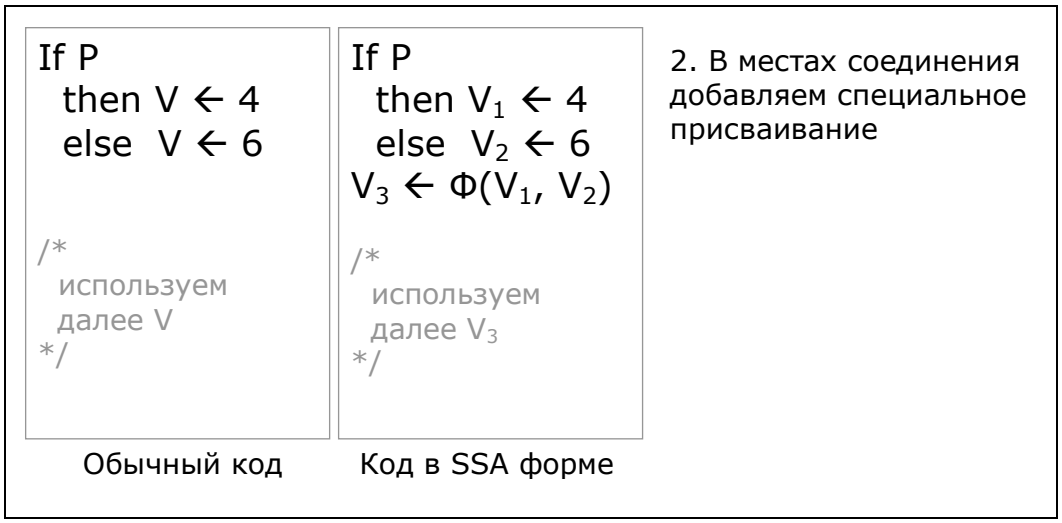


Рисунок 8

Управляющий граф

Предложения программы организуются в *базовые блоки* (не обязательно максимальные). Базовым блоком может быть любой фрагмент кода не содержащий переходов. Программа входит в первое предложение такого блока и выходит из последнего.

Управляющий граф (control flow graph) – это направленный граф, в вершинах которого расположены базовые блоки, а дуги соответствуют переходам.

```
1:  I ← 1; J ← 1;
1:  K ← 1; L ← 1;
2:  repeat
2:        if (P)
3:           then do
3:              J ← I
3:              if (Q)
4:                    then L ← 2
5:                    else L ← 3
6:              K ← K + 1
6:           end
7:           else K ← K + 2
8:        print(I, J, K, L)
9:        repeat
9:           if (R)
10:             then L ← L + 4
11:       until (S)
12:       I ← I + 6
12: until (T)
```

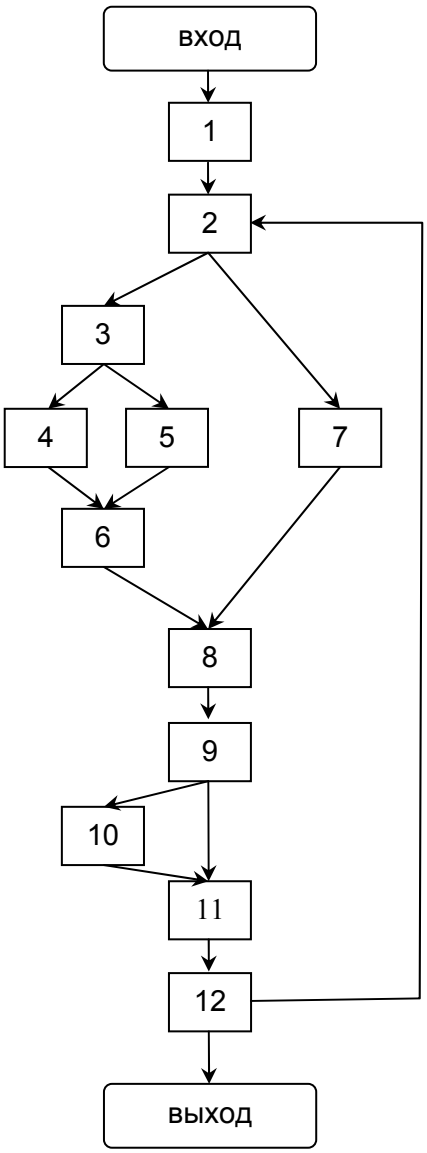


Рисунок 10

Преобразование в SSA

Пусть программа представлена в виде control flow graph.

Каждое предложение внутреннего представления вычисляет некоторое выражение и использует результат для присваивания или перехода.

Преобразование программы в SSA форму – двухэтапный процесс:

1. На первом этапе добавляются тривиальные *Ф*-функции в некоторые вершины графа управляющей логики.

2. На втором этапе генерируются новые переменные (находятся зависимости, переменные получают «версии»).

Массивы

«Хитрость» работы с массивами заключается в том, что все обращения к элементам массивом оборачиваются специальными функциями.

Исключение составляет тот случай, если язык поддерживает операции с массивами как со скалярами (поэлементное копирование и пр.). В этом случае переменная массива обрабатывается как обычный скаляр.

Рассмотрим случай обращения к элементу массива.

Исходный код, использующий массивы:

```
      ←  A(i)
A(j)  ←  A(i)
      ←  A(k) + 2
```

Эквивалентный код, в котором использованы специальные операторы доступа:

```
      ←  Access(A, i)
A(j)  ←  Update(A, j, V)
T     ←  Access(A, k)
      ←  T + 2
```

SSA форма:

```
      ←  Access(A8, i7)
A9(j) ←  Update(A8, j6, V5)
T1    ←  Access(A9, k4)
      ←  T1 + 2
```

Применение SSA

- Продвижение констант
- Удаление «мертвого кода»
- SSA представление также используется для определения эквивалентности программ.

Пример продвижения констант и удаления «мёртвого кода» представлен в Таблице 3.

Таблица 3

Оригинальный код	Код после продвижения констант	Код после удаления «мёртвого кода».

a1 = 10;
b1 = 3;
c1 = a1 + b1;
if(c1 < V1)
  a2 = a1 + 3;
else
  a3 = b1 + 10;
a4 = Φ(a2, a3);
c2 = a4 – b1;
printf("%d\n", c2);
	a1 = 10;
	b1 = 3;
	c1 = 13;
	if(13 < V1)
	  a2 = 13;
	else
	  a3 = 13;
	a4 = Φ(a2, a3);
	c2 = 10;
	printf("%d\n", 10);
		printf("%d\n", 10);

Анализ исходной программы

Анализ исходной программы на языке высокого уровня обычно состоит из трех логических этапов:

- *Лексический анализ* – линейный анализ, при котором поток символов исходной программы считывается слева направо и группируется в токены (token), представляющие собой последовательности символов с определенным совокупным значением.
- *Синтаксический анализ* – иерархический анализ, при котором символы или токены иерархически группируются во вложенные конструкции с совокупным значением.
- *Семантический анализ* – позволяет проверить корректность совместного размещения компонентов программы (например, типы данных).

Языки

Т. Пратт в книге «Языки программирования» приводит перечень свойств хорошего языка:

- Ясность, простота и единообразие понятий языка
- Ортогональность
- Естественность для приложений
- Поддержка абстракций
- Удобство верификации программы
- Среда программирования
- Переносимость программ
- Стоимость использования

- Стоимость выполнения программы
- Стоимость трансляции программы
- Стоимость создания, тестирования и использования программы
- Стоимость сопровождения программы

Разнообразие предметных областей и желание приблизить язык по уровню абстракций и естественности представления алгоритма к каждой области порождает новые и новые языки. Существенным компонентом оценки каждого решения является стоимость, которая складывается из если не взаимоисключающих, то, по меньшей мере, несовместимых компонент. Конкретный выбор в пользу того или иного критерия делается в зависимости от сферы применения компилятора. Так для компиляции большинства студенческих программ важно скорость компиляции и несущественна скорость исполнения. Скорость компиляции важна и в JIT-компиляторах.

Принятая модель классификации языков выделяет четыре основные вычислительные модели:

- *Императивные (процедурные) языки.* Состояние машины – память. Программа – последовательность операторов. Исполнение оператора влечет изменения состояния памяти. Примеры: C, C++, FORTRAN, ALGOL, PL/1, Pascal, Ada, Smalltalk, COBOL.
- *Аппликативные (функциональные) языки.* Начальное состояние машины – память. Программа – композиция функций. Примеры: ML, Lisp.
- *Языки, основанные на системе правил (логического программирования).* Программа – совокупность пар (разрешающее условие, действие). Примеры: Prolog, НБФ в YACC, Lex.

- *Объектно-ориентированное программирование.* В этой модели строятся сложные объекты данных и определяются операции над этими объектами.

На язык оказывает влияние и среда программирования. Преимущественно на возможности языка, упрощающие раздельную компиляцию и сборку программы (модульность), и на возможности тестирования и отладки программы.

Архитектура также оказывает влияние на языки. В основном причиной модификации языка для архитектуры служит желание значительно повысить производительность конечной программы. Пример: OpenMP расширение.

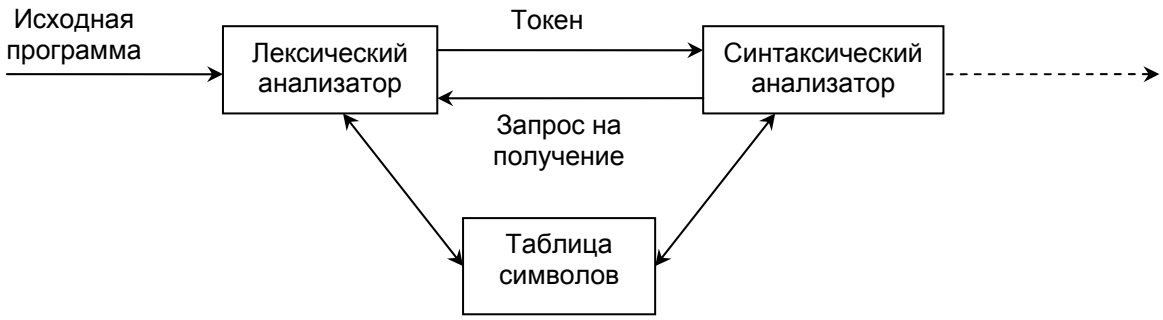
Ключевым вопросом в реализации языка программирования является то, какое представление имеет программа во время её выполнения на реальном компьютере. В зависимости от реализации языка их делят на:

- *Компилируемые.* Языки C, C++, FORTRAN, Pascal, Ada принято считать компилируемыми. Транслятор компилируемого языка обычно является довольно большой и сложной программой, и максимальное значение имеет создание максимально эффективных с точки зрения исполнения программ.
- *Интерпретируемые.* Языки LISP, ML, Perl, Postscript, Prolog и Smalltalk обычно реализуются с помощью интерпретаторов. Трансляторы таких языков обычно – сравнительно простые программы. Основная сложность заключается в реализации интерпретатора.

Лексический и синтаксический анализы

Задача лексического анализатора состоит в чтении потока символов и выдачи потока токенов. Синтаксический анализатор получает строку

токенов от лексического анализатора и проверяет, может ли эта строка порождаться грамматикой исходного языка.



Для построения лексических анализаторов на основе спецификаций, использующих регулярные выражения, был создан ряд специальных программных инструментов. В приложении описан инструмент под названием Lex.

Для построения синтаксических анализаторов на основе спецификаций в приложении рассматривается генератор LALR-анализаторов Yacc.

Front end

В GCC от языка зависит только *Front end* (часть компилятора, анализирующая программу на языке высокого уровня и преобразующая её во внутреннее представление программы в компиляторе).

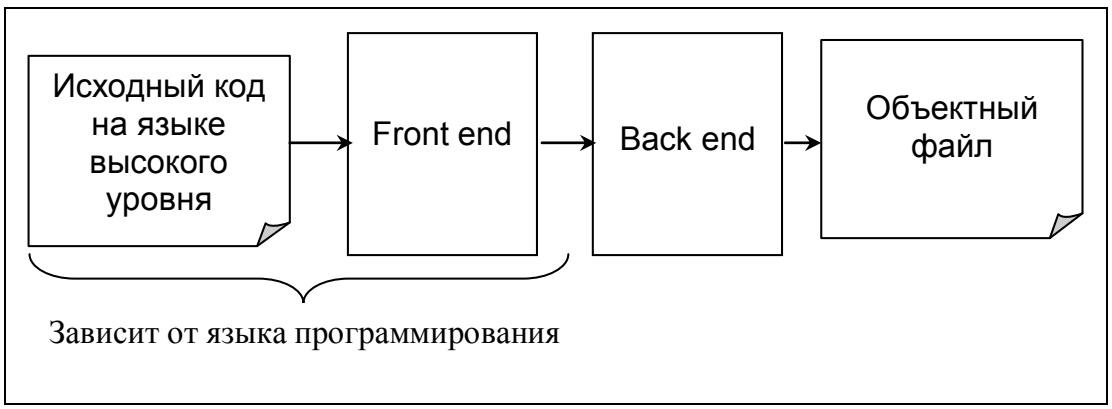


Рисунок 11

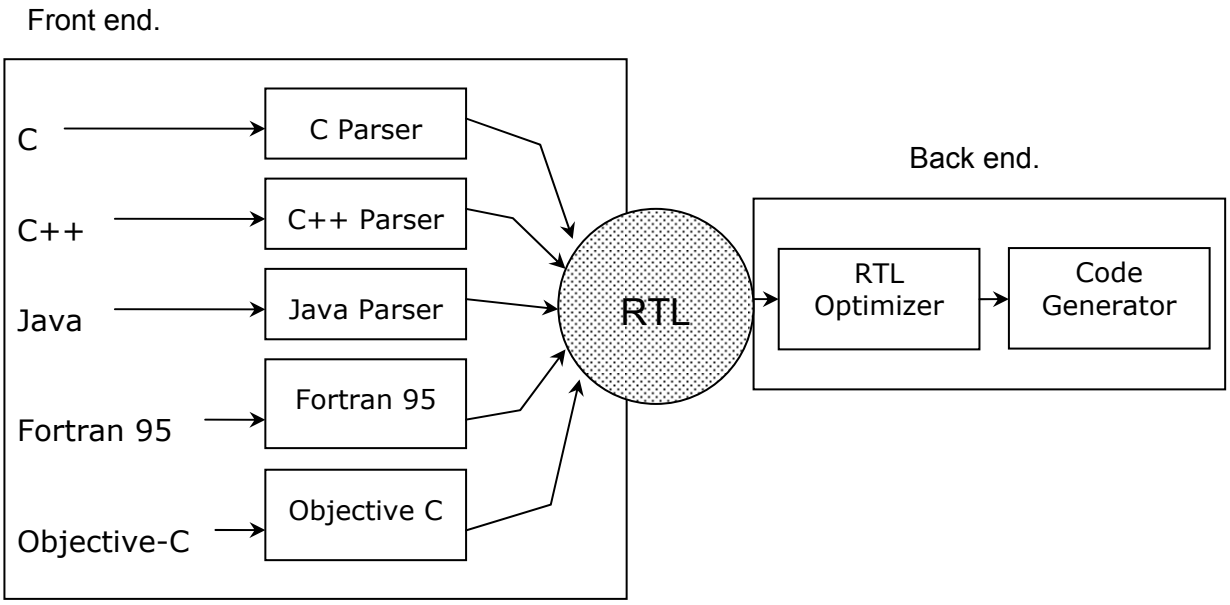


Рисунок 12

Front end должен обеспечивать: лексический анализ, синтаксический анализ, генерацию кода во внутреннем представлении. Также front end может включать предварительную оптимизацию: например, вычисление константных выражений, упрощение арифметических выражений.

Создание компилятора для нового языка в GCC означает создание нового front end-а.

Обычно front end работает по схеме показанной на рисунке 13.

Обрабатывая очередную лексему средствами функций предоставленных GCC, формируется дерево, хранящее информацию о распознанном участке программы.

Центральной структурой данных в Front end является дерево (по форме очень похоже на дерево синтаксического разбора). Узлы дерева способны хранить целые или вещественные числа, строки, операторы и пр. В общем случае дерево не является универсальным для всех front end-ов.

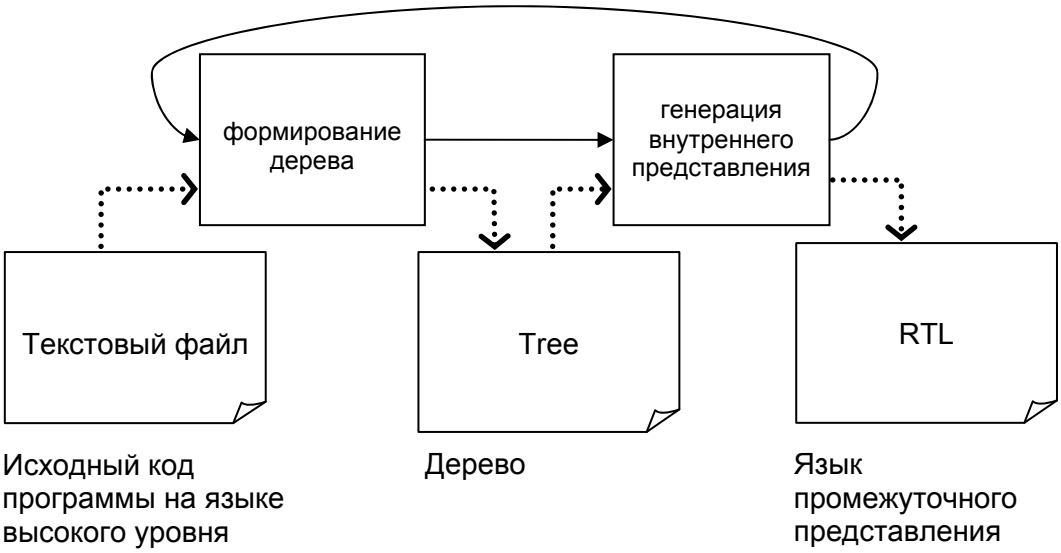


Рисунок 13

Как только завершается анализ осмысленного участка кода, генерируется фрагмент RTL кода на основании сформированного дерева.

Создание собственного Front end

В данной главе описывается процесс создания игрушечного front end-а VMK.

Создание собственного Front end достаточно хорошо описано в документе GCC Front end HOWTO от Sreejith K Menon. Если Вы собираетесь начать писать собственный Front end, то, несомненно, следует начать с прочтения этого документа.

Имя каталога содержащего файлы с реализацией front end совпадает с именем языка и располагается в папке gcc.

Как правило, никто не создает front end с нуля. Для этого с GCC идет стандартный пример treelang. Нужно скопировать его в свой каталог и начать модифицировать.

Файлы, которые получились в результате первого этапа (копирования стандартного front end-а и смены имен) представлены в Таблице 4.

Таблица 4

Файл	Назначение
ChangeLog	История вашего font end'а. Файл необязателен, но этикет требует его присутствия.
Make-lang.in	Включается в главный makefile в каталоге gcc. Его главная роль – вызвать makefile в каталоге языка. Обязательный файл.
Makefile.in	Используется для создания makefile в директории языка. Обязательный файл.
config-lang.in	Обрабатывается скриптом configure.
lang-specs.h	Содержит информацию для программы gcc о новом компиляторе (расширения файлов, возможные параметры).

Файлы настройки на этом закончились. Теперь осталась реализация самого front end. Для построения минимального варианта понадобится реализация следующих модулей, описанных в Таблице 5.

Таблица 5

Файл	Назначение
Lex.l	Описания токенов для лексического анализатора на входном языке LEX.
Parce.y	Описание синтаксиса для YACC. Генерация дерева внутреннего представления и генерация RTL.
Vmk.c, vmk.h	Собственно реализация компилятора. Инициализации... А также заглушки для функций, которые от нас требует gcc.
Vmk1.c	Сюда по примеру treelang вынесены callback функции инициализации компилятора, парсера, открытия/закрытия файлов, декодирования параметров командной строки. В известном примере TOY было всё в одном файле, но мы решили его не загромождать.

Язык.

Разработанный front end воспринимает язык, который позволяет в текстовой формулировке записывать унарные, бинарные и тернарные математические функции.

Ограничения языка:

- оперирует целыми числами;
- четыре математических операции (плюс, минус, умножить, делить).

Пример:

```
binary function a is
    firstly first argument plus second then mul ten;
```

Что эквивалентно следующей программе на языке Си:

```
a( int first, int second )
{ return ( first + second ) * 10 ; }
```

Исходные материалы.

Наиболее подробно процесс создание Front end описан в документе GCC Front end HOWTO от Sreejith K Menon

Для очень старых версий GCC существовал Front end TOY, который фактически реализовывал нужную нам функциональность

В коллекции GCC существует демонстрационный язык treelang – пример того, как нужно создавать front end-ы. Но он гораздо крупнее TOY и не обладает нужной прозрачностью для лабораторной работы.

Реализация front end-а для языка, функциональность которого не превышает Си, сводится к вызовам нужных API функции работы с деревом и RTL. В том же случае, если Вы реализуете font end для более сложного языка, то Вам предстоит свести все дополнительные возможности к имеющимся примитивам.

В нашем случае всё очень просто. Приведённые в приложении два файла `lex.l` и `parce.y` полностью описывают язык.

Генерация кода

Процесс генерации кода для базового блока (ББ) состоит из трёх основных этапов:

1. выбор инструкций целевого МП – сопоставление некоторой операции выражения соответствующей инструкции, перед сопоставлением инструкций производится определение класса регистров для имеющихся в программе переменных и промежуточных значений;
2. распределение регистров – размещение переменных в регистрах самым эффективным образом;
3. составление расписания инструкций – формирование машинного кода базового блока из выбранных инструкций с наименьшим временем исполнения.

Первый этап генерации кода — сопоставления графа зависимостей по данным и управлению программы инструкциям целевого МП. Для сопоставления инструкций базовый блок представляется в виде ациклического сильно связанного двудольного орграфа потока данных (орграф ББ):

$K_b = (V, E)$,

где $V = \{v_i\}$ – множество вершин двух типов:

а) виртуальные (только с обозначением регистрового субфайла, а не номера, поскольку регистры в регионе функционально эквивалентны) регистры;

б) команды;

E – множество ориентированных рёбер, представляющих связи по данным,

$$E = \{e_k\},\ e_k = (v_i, v_j).$$

На орграфе K_b вводится функция связности $f_c = V \times V \in \{0, 1\}$, и функция разметки вершин $f_T(v) \in \{"ресурс", "команда"\}$, $v \in V$. Вводится обязательное условие: если $f_c(v_i, v_j) = 1$, то $f_T(v_i) \neq f_T(v_j)$.

Для каждой вершины $v \in V$ определяется имя ресурса или команды, и вводятся атрибуты вершины. Для $\forall v \in V$, $f_T(v) = "команда"$, вводится множество операндов $IN(v)$: $v_i \in IN(v)$, при $f_T(v_i) = "ресурс"$, и существует $e_k = (v_i, v)$; и множество результатов $OUT(v)$: $v_j \in OUT(v)$, при $f_T(v_j) = "ресурс"$, и существует $e_m = (v, v_j)$. При таком представлении ББ автоматически выделяется суперскалярный параллелизм. В орграфе ББ выделяется множество начальных вершин (либо вводится фиктивная начальная вершина-команда) и проводится топологическая сортировка для подготовки этапа выбора инструкций.

Выбор инструкций

Выбор инструкций для сопоставления с орграфом ББ происходит в процессе полного покрытия орграфа ББ необходимым количеством копий графов инструкций МП. Процесс сопоставления является итеративным и может происходить одновременно с окончательным распределением регистров и построением расписания инструкций (генерацией кода). В процессе сопоставления каждой исходной переменной $var_i \in VAR$ определяется её размещение в памяти и множество классов регистров для размещения копий переменной.

После сопоставления каждой переменной множества классов регистров каждой операции ББ сопоставляются одна (в улучшенных кодогенераторах все возможные) инструкции МП. Например, при существовании разных

инструкций сложения: сложения любых регистров (арифметических, индексных, и т.п.) и сложения регистров из некоторого субфайла, то исходной операции сложения сопоставляются либо наиболее «приоритетная», либо первая встретившаяся в описании (GCC), либо все возможные инструкции. Из сопоставленных инструкций при составлении расписания команд выбирается только одна оптимальная.

Для некоторого ББ $B=(V_B,E_B)$ для $\forall v\in V_B$, $f_T(v)="команда"$, сопоставленными являются все $Instr_I$, для которых в орграфе инструкции МП $K_G=(V_K,E_K)$ существует команда $v'\in V_K$, $f_T(v')="команда"$, такая, что совпадают виртуальные имена (т.е. «+», «-»), $\|IN(v)\|=\|IN(v')\|$, $\|OUT(v)\|=\|OUT(v')\|$, $IN(v')_i\in RT(TRT(Type(IN(v)_i)))$ для каждого $i=1,2,\ldots,\|IN(v)\|$, и $OUT(v')_i\in RT(TRT(Type(OUT(v)_i)))$ для каждого $j=1,2,\ldots,\|OUT(v)\|$ – т.е. инструкция МП может быть сопоставлена только в случае, если типы операндов и результата исходной операции и инструкции совпадают. В случае комбинированных команд (исполняющих несколько операций над данными за один такт) используется расширенное сопоставление. Комбинированной команде с орграфом $K_G=(V_K,E_K)$ инструкциями $\{ v'_1,\ldots,v'_n \} \mid v'_i\in V_K$, $f_T(v'_i)="команда"$ сопоставляются $v_1,\ldots,v_n \mid v_i\in V_B$, $f_T(v)="команда"$, с совпадающими виртуальными именами. Для $i=1,\ldots,n$: $\|IN(v_i)\|=\|IN(v_i')\|$, $\|OUT(v_i)\|=\|OUT(v'_i)\|$, для $j=1,2,\ldots,\|IN(v)\|$: $IN(v')_j\in RT(TRT((Type(IN(v)_j))))$, и для $k=1,2,\ldots,\|OUT(v)\|$: $OUT(v')_k\in RT(TRT(Type(OUT(v)_k)))$. Для представления структуры комбинированной команды используются следующие ограничения: для некоторых v_i и v_j, если $IN(v_i)\equiv IN(v_j)$ или $IN(v_i)\equiv OUT(v_j)$, необходимо, чтобы для v'_i и v'_j, сопоставленных соответственно v_i и v_j, было справедливо $OUT(v'_i)\equiv IN(v'_j)$ или $IN(v'_i)\equiv OUT(v'_j)$.

Сопоставления выполняется с вершин v_i, имеющих $IN(v_i)=\varnothing$ (начальные вершины орграфа ББ) в направлении ориентации дуг орграфа. В GCC

сопоставление комбинированных команд происходит в отдельном проходе. В некоторых компиляторах приоритет при сопоставлении имеют комбинированные команды. Специальные команды, состоящие из нескольких операций (например, выборка из памяти с автоинкрементном) формируются либо в отдельном проходе, либо далее в процессе составления расписания команд.

Распределение регистров.

Практически важнейшей проблемой при генерации коду является распределение регистров, основанное на определении времени жизни переменных при выполнении кода функции. Большинство арифметико-логических команд МП с RISC архитектурой не могут оперировать непосредственно со значениями в памяти, поэтому компилятор генерирует последовательность команд для загрузки необходимых в некоторый момент времени значений переменных в регистры. Для определения необходимых на текущий момент значений в регистровом файле необходимо частично разрешить задачу определения времени жизни переменных, копии которых находятся в регистровом файле (регистровых переменных).

В современных моделях МП с длинным командным словом, улучшенных RISC-МП, процессоров с архитектурой EPIC, нетрадиционных параллельных процессоров, ПЦОС ёмкость регистрового файла достигает 64 – 256 регистров. Алгоритм распределения регистров должен эффективно задействовать такое количество регистров.

В случае исполнения одной команды за такт (типичный RISC) особых проблем не возникает. Сначала строится расписание арифметико-логических команд, затем между командами могут быть вставлены (при необходимости) дополнительные команды загрузки или сохранения

регистров в память. При более сложной архитектуре процессора, установка дополнительных команд проводилась итеративно до определения оптимального варианта, а затем все команды участвовали в составлении расписания.

При количестве одновременно исполняемых операций в длинной команде от 4 до 8 (или больше) проблема распределения регистров становится гораздо более сложной. Обычно это связано с тем, что возможности подачи данных из оперативной памяти в регистровый файл ограничены. Несколько легче эта проблема в случае, если мы обрабатываем регулярные большие куски данных – например, массивы, но, если доступ требуется к нескольким переменным, расположенным в памяти далеко друг от друга, проблема становится очевидной – вычислительные возможности процессора простаивают, так как скорость подачи данных относительно невелика. Решение этой проблемы путём увеличения количества интерфейсов к оперативной памяти частично решает проблему подачи данных (гарвардская и супергарвардская архитектуры), но компиляция становится ещё более сложной задачей. Естественно, существует некоторое количество неочевидных оптимизаций, связанных с группированием данных и загрузкой векторов, но такие средства – редкость.

Для старых моделей МП класса Intel 80486 проблема скорости подачи данных не поднималась в принципе, потому что ёмкость активной части регистрового файла не превышала 6 регистров, кроме того, инструкции МП обычно прямо оперировали значениями, расположенными в памяти. Однако, с появлением суперскалярных архитектур, проблема подачи данных компенсировалась в основном кэшем, и, в общем, анализ проводился с учётом того, что данные уже находятся в кэше 1-го уровня. Если рассмотреть процессор Itanium, то становится понятным, что

необходимы серьёзные меры для оптимального использования 128 целочисленных регистров и 128 регистров для чисел с плавающей точкой.

Среди алгоритмов распределения регистров рассмотрим два метода, широко использующихся на практике: распределение регистров с помощью раскраски графов и распределением регистров с помощью линейного сканирования потока данных.

Оба метода используют т.н. информацию о времени жизни переменных для нахождения распределения переменных-кандидатов на загрузку в регистры на машинные регистры.

Для достижения цели метод, связанный с раскраской графов, представляет информацию о времени жизни в виде графа зависимостей, в котором вершины представляют собой переменные-кандидаты и между двумя вершинами существует дуга, если их времена жизни пересекаются – то есть эти переменные не могут содержаться в одном регистре. Для целевого процессора, содержащего N регистров, нахождение раскраски этого графа зависимостей времени жизни в N цветов эквивалентно распределению переменных-кандидатов по регистрам без конфликтов. Первым описание процесса раскраски графа зависимостей времени жизни встречается у Чайтина (Chaitin), этим вопросам посвящена диссертационная работа Бриггса (Briggs) [27]. Алгоритм итеративно строит граф зависимостей и пытается его раскрасить. Если раскраска не удаётся, некоторые кандидаты перемещаются в память, убирается часть связей в графе, вставляется код для выгрузки/загрузки переменных (*spill code*), и процесс повторяется. На практике основное время уходит на процесс построения графа, размер которого является $O(n^2)$ от количества переменных-кандидатов. Так как модуль программы может иметь сотни переменных, и компилятор сам может генерировать достаточное количество временных переменных в

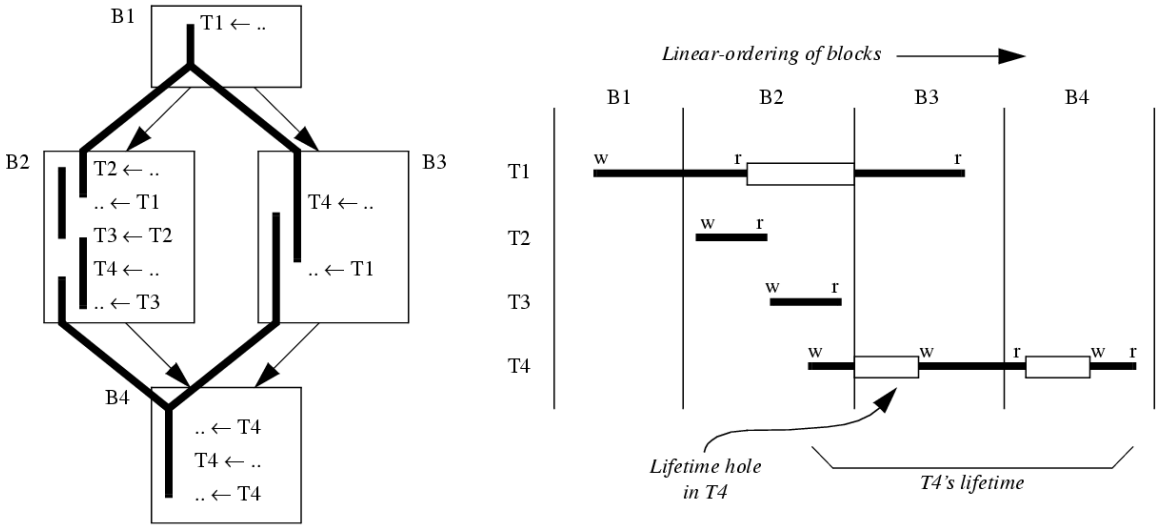


Рисунок 14

случае агрессивных оптимизаций, раскраска графа может занять длительное время.

Распределение регистров с помощью сканирования потока данных оперирует с понятием «времени жизни», который начинается в момент присвоения переменной этого значения и заканчивается в момент последнего использования этого значения в некоторой ветви программы. При сканировании просматривается в линейном порядке код программы, с целью определения количества «активных» переменных. Количество «активных» интервалов определяет количество необходимых регистров в данной точке программы. В случае если их слишком много, ряд переменных временно сохраняется из регистров в память. В принципе, время работы такого распределителя линейно.

Так как последний алгоритм весьма прямолинеен, используется ряд улучшений, рассмотрим их.

При анализе времени жизни полезно выделить места, где переменная не имеет полезного значения. Например, на рисунке 14 показан базовый блок и обозначено время жизни переменных с указанием *дыр* во времени жизни.

В частности, если мы распределяем регистр *r* для переменной *t*, но в её времени жизни есть дыра, в которую умещается время жизни другой переменной *u*, то мы можем распределить переменной *u* тот же регистр *r*. В частности, на рисунке переменная *T3* целиком помещается в дыру во времени жизни *T1*.

Для подсчета времён жизни и «дыр» в них необходим реверсный проход по программному коду.

Таким образом, регистры представляются в виде ячеек, каждой из которых в некоторый момент времени может соответствовать только одно значение. Для того, чтобы несколько переменных могли быть размещены в ячейке необходимо, чтобы их времена жизни не пересекались. Если говорить о дырах во времени жизни, то две переменные также можно разместить в одном регистре, если время жизни одной из них целиком умещается в дыре во времени жизни другой. Эта трактовка несколько неоднозначна – фактически дыра во времени жизни обозначает, что образована новая переменная, но здесь понятие «дыры» вводится для удобства представления.

Таким образом, распределитель просматривает программу в прямом направлении, собирая информацию о времени жизни. При необходимости назначения временной переменной *t* регистра выбирается либо один из свободных регистров, либо же выбирается регистр, содержащий переменную, у которой сейчас существует «дыра» и время жизни *t* целиком умещается в «дыре». После сопоставления регистра все обращения к *t* в программе заменяются обращениями к *r*.

Если свободных регистров нет, необходимо подыскать «занятый» регистр, освобождение которого будет наименее болезненно. Должна быть выбрана переменная, которая будет позже остальных востребована, при этом необходимо учитывать глубину вложенности цикла, где она востребуется.

Фактически, все переменные сохраняются в регистрах как можно дольше и записываются в память только в том случае, если регистр нужен для «более приоритетной» переменной. Ещё одна оптимизация заключается в том, что регистры, значение переменной в которых не отличается от значения в памяти, не будут записываться обратно в память.

При распределении регистров необходимо решить ещё одну проблему. В случае, если переменная вытесняется из памяти в одном или обоих базовых блоках B2 и/или B3, к моменту перехода потока управления в блок B4 фактически две копии переменной могут находиться в разных регистрах, либо же в одном потоке переменная может находиться в регистре, а в другом – в памяти. В этом случае производятся следующие действия: 1) если копии переменной находятся в разных регистрах, то в один из потоков управления вставляется команда копирования из регистра в регистр. В случае, если переменная в одном потоке управления изменена, а в другом её нет в регистрах, делается сохранение переменной в памяти. Вообще, поведение распределителя в данном случае может зависеть от обстоятельств – если эта переменная будет использована практически сразу после слияния потоков управления, есть смысл загрузить её в регистр в той ветви, гдё её нет в регистре до слияния потоков.

Более гибкое решение предполагает, что регистры у нас равноправны, следовательно, привязка регистров может быть нежёсткой, чтобы затем можно было подкорректировать номера при слиянии потоков, что позволит сэкономить несколько команд, но с другой стороны, усложнит алгоритм.

Ещё несколько улучшений алгоритма касаются поддержки современных микропроцессоров. Для процессоров с длинным командным словом алгоритмы усложнены – так как у нас несколько времён жизни переменных могут начинаться и завершаться одновременно, кроме того,

команда загрузки/сохранения обычно может быть только одна на несколько параллельно исполняющихся арифметических команд. В этом случае переменные должны загружаться в память значительно раньше места их затребования, так как одна длинная команда может потребовать более трёх-четырёх переменных.

Подытоживая, отметим, что в принципе алгоритмы распределения регистров достаточно просты и понятны. С другой стороны архитектура процессора оказывает определяющее влияние, и алгоритм обрастает достаточно сложными эвристиками.

Генерация кода

Генерация кода может выполняться как минимум двумя общими способами: 1) полный перебор; 2) эвристические методы, основанные на списочных расписаниях. Полный перебор обеспечивает оптимальный код, но из-за его слишком долгого времени (функция экспоненциальна) исполнения, он не может быть использован даже на самых быстрых ПЭВМ. Эвристические методы обеспечивают генерацию кода за квазилинейное время, но могут давать погрешность от 5% до 15%.

Для орграфа ББ $K_b=(V_b,E_b)$ вводится фиктивная начальная вершина v_0, $f_T(v_0)="команда"$, и дуги $e=(v_0,v_i)$ для всех v_i, для которых $f_T(v_i)="ресурс"$, и не существует такого $v\in V$, что $f_T(v)="команда"$, $v_i\in OUT(v)$. Аналогично вводим конечную вершину v_E, $f_T(v_E)="команда"$, и дуги $e=(v_j,v_E)$, для всех v_j, для которых $f_T(v_j)="ресурс"$, и не существует такого $v\in V$, что $f_T(v)="команда"$, $v_j\in IN(v)$. Для K_b вводятся следующие метрики:

1) λ_v – длина пути из начальной вершины v_0 в вершину v;
2) γ_v – длина пути из вершины v в конечную вершину v_E;
3) l_{ij} – длина пути из вершины v_i в вершину v_j.

Примем, что для $e_k=(v_i,v_j)$, где $f_T(v_j)="команда"$, $l_{ij}=0$, а для $e_m=(v_i,v_j)$, где $f_T(v_j)="ресурс"$, l_{ij} определяется временем исполнения команды v_j. Дополнительно вводим метрику – длину всего K_b Λ_b – длина пути из вершины v_0 в v_E.

Метрика Λ_b фактически является временем исполнения ББ на МП с неограниченными ресурсами. В случае ограничения по ресурсам, реальное время исполнения вырастает обратно пропорционально к степени поддержки МП скалярного параллелизма.

В процессе генерации кода ББ вводится время t, которое при генерации кода для начальной вершины ББ равняется 0, и увеличивается на единицу с каждой сгенерированной командой базового блока. Для команды v_i ($f_T(v_i)="команда"$), исполнение которой началось в момент времени t_0, результат будет получен в регистрах $v_j(f_T(v_j)="ресурс"$) в момент времени t_0+dt, где dt – время исполнения v_i (для системы команд RISC/CISC). Условием исполнения команды является доступность необходимый ей регистровых операндов в регистрах и освобождение необходимых инструкции конвейеров, что проверяется с помощью конечного автомата, представляющего конвейер. В цикле генерации команды для $K_B=(V_B,E_B)$ просматриваются готовые к исполнению команды $v_i \in V$, где для всех команд операнды присутствуют в физических регистрах. С сопоставленных операциям ББ инструкций МП определяется множество $V = \{ v_i \}$, где v_i – готовая к исполнению команда. Исходя из множества V определяется итоговая команда K_i, для которой функция оценки максимальная. В качестве функции оценки используется сумма путей (для одной команды – просто значение пути) атомарных команд-компонент сформированной команды $cbest(V)=\Sigma\gamma_I$, где γ_i – путь до конечной вершины для некоторой рассматриваемой команды. На формирование множества V оказывает влияние информация о наличии свободных конвейеров

функциональных устройств, обновляющаяся после каждой генерации команды в каждый момент времени t согласно описаниям, скомпилированным в конечный автомат. В случае конфликта из-за занятости ресурса используется, например, алгоритм, описанный в [15], проиллюстрированный на следующем рисунке:

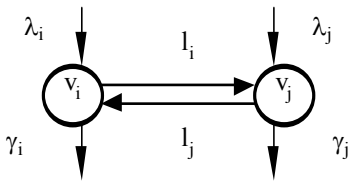


Рисунок 15. Конфликт между командами по использованию функционального устройства.

Как l_i и l_j обозначены времена исполнения команд v_i и v_j. Для определения очерёдности при исполнении команд вычисляется значение логического выражения $\lambda_i+l_i+\gamma_j \geq \lambda_j+l_j+\gamma_i$. При истинности выражения исполняется v_j, иначе – v_i, с целью минимального увеличения высоты Λ_b графа ББ. Такой же алгоритм используется при рассмотрении конфликтов по использованию полей длинной команды.

Алгоритм списочных расписаний, основанный на алгоритме поиска кратчайшего пути в графе, позволяет получить практически оптимальные результаты [11]. Для оптимизации использования алгоритма для МП со сложной архитектурой, обычно длинным командным словом, и большим регистровым файлом, построение списочного расписания интегрируется с распределением регистров.

Программная конвейеризация.

Для эффективной генерации кода цикла и анализа качества процесса генерации кода необходимо иметь возможность нахождения максимально возможного параллелизма для конкретного ярда алгоритма. Приведём пример: процессоры с длинным командным словом могут исполнять за такт около 4-8 предварительно определённых компилятором команд. Если

учесть, что в последовательной программе степень скалярного параллелизма не превышает 2 (т.е. обычно не более 2-х команд могут выполняться одновременно), возникает вопрос о том, как же задействовать имеющиеся у нас в распоряжении вычислительные мощности, если в среднем степень параллелизма настолько мала? Ответ на этот вопрос нетривиален – процессоры, которые могут выполнять за такт большое количество арифметико-логических операций, рассчитаны на исполнение циклов. При этом вид исполнения предполагается такой: поскольку направление перехода в цикле обычно известно (всё время на начало), можно считать, что за телом цикла следует такое же тело (но представляет собой следующую итерацию). Таким образом поступают современные суперскалярные процессоры: они могут просматривать не только первую итерацию, но и вторую и третью и т.д., представляя их в виде линейного участка, при этом может быть начато исполнение готовых команд из любой итерации. Таким образом, исполнение последовательных итераций цикла на самом деле может перекрываться. В случае процессора с длинным командным словом исполнение этой задачи берёт на себя компилятор, в этом случае выполнение цикла с перекрытием итераций будет называться «программно конвейеризированным», а метод получения такого расписания – программной конвейеризацией. При этом среднее время, проходящее между началом выполнения последовательных итераций называется *интервалом инициации итераций* (ИИИ).

Существует достаточно много алгоритмов программной конвейеризации. Можно выделить версии «модульного планирования», где первоначальный ИИИ равен времени исполнения неконвейеризированного цикла, а затем постепенно уменьшается на единицу. В цикле производятся попытки построения реалистичного расписания команд. Расписание для минимально возможного ИИИ считается окончательным. Обычно этот алгоритм не гарантирует нахождение наилучшего решения.

В ряде случаев используются другие алгоритмы, например углубленное конвейерно-проникающее планирование потока команд [16] – алгоритм EPS (Enhanced Pipeline Scheduling) К. Эбчоглу [31], оперирующий с ациклическим графом базового блока.

Алгоритм EPS не похож на алгоритмы программной конвейеризации, основанные на модульном планировании и выделении ядра. Алгоритм использует оригинальный подход к программной конвейеризации, основанный на перемещении кода с условием сохранения структуры тела цикла [31]. Алгоритм очень похож на схему распараллеливания циклов, применяемую в суперскалярных процессорах. Основным недостатком этого алгоритма, ограничивающего его применение в некоторых условиях, является ориентированность на неограниченные ресурсы МП и неограниченный скалярный параллелизм. Алгоритм EPS состоит из двух этапов: 1) глобальное перемещение кода с переименованием и подстановкой вперёд; 2) конвейеризация тела цикла.

1. Глобальное перемещение кода с переименованием и подстановкой вперёд используется для перемещения операции, которая находится после условного оператора, вперёд этого оператора для укорочения антизависимостей. Переименования превращает операцию $x = y\ op\ z$ в две операции: $x' = y\ op\ z;\ x=x'$. Первое присвоение определяет переменную, которая используется только для операции копирования $x=x'$, поэтому её можно вынести вперёд условного оператора. Например, код:

```
if (a>0) x=y+z;
```

с помощью этого метода превращается в:

```
x0=y+z;     if (a>0) x=x0;
```

Вынесенный вперёд условного оператора оператор называется *спекулятивно исполняемым* – потому что его исполнение необходимо только в случае истинности условия условного оператора. Спекулятивное

исполнение широко используется в разных методах оптимизации кода для микропроцессоров, поддерживающих скалярный параллелизм [40].

Для оператора присвоения *var=expr подстановкой вперёд* называется изменение использования *var* в следующих за присвоением операторах на *expr*. Последнее полезно если в результате подстановки вперёд операторы смогут исполняться параллельно. Обе операции проиллюстрированы на следующем рисунке:

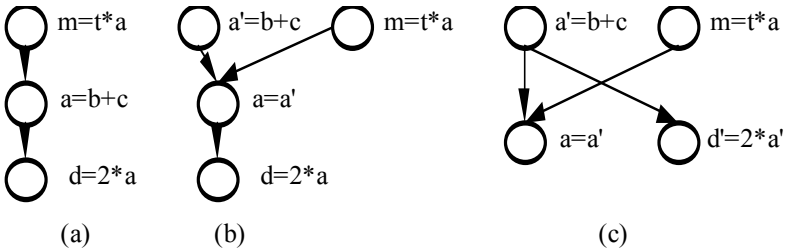


Рисунок 16. Пример подстановки вперёд и переименования переменных.

На рисунке (a) изображён граф зависимостей по данным до преобразования, на (b) – граф зависимостей после переименования *a=b+c*, на (c) – после подстановки вперёд *a'*. Дополнительно подстановка вперёд может разрушать прямые зависимости, мешающие перемещению кода. Зависимость $S_1 \delta S_2$ может быть разрушена подстановкой вперёд, если S_1 – операция копирования, или S_1 и S_2 имеют в качестве операндов константы. Последовательность операторов

S1: x = z + 4; S2: y = x + 2; c S1δS2

заменяется последовательностью

S1: x = z + 4; S2: y = z + 6;

где S1 и S2 могут исполняться параллельно.

При конвейеризации цикла операторы перемещаются против дуг зависимостей по управлению. Алгоритм конвейеризации содержит в себе две фазы, которые итеративно повторяются пока операторы ещё имеют возможность перемещения, или при генерации расписания команд цикла

начинают циклически повторяться инструкции последней генерированной команды. Во время первой фазы операторы тела цикла перемещаются вперёд на сколько это позволяют зависимости по данным и управлению. Первые инструкции тела цикла, исполняющиеся параллельно, называются «*границей*» – операторы цикла перемещаются вперёд любым образом, но не вперёд границы — она ограничивает перемещение кода. Во время другой фазы инструкции, стоящие «на границе» дублируются и перемещаются. Операторы дублируются при перемещении через верхнюю «границу» цикла, потому что каждый из них имеет двух потомков по управлению. Операторы, выносящиеся вперёд границы, формируют пролог цикла. Дублирующие операторы добавляются к этому же телу циклу в конец, и обозначаются как код из следующей итерации. Этот алгоритм используется итеративно, пока операторы будут иметь возможность перемещения или процесс генерации расписания инструкций не зациклится. Например, для алгоритма цифровой фильтрации

```
for(i=0;i<N;i++) s+=coef[i]*data[i];
```

после проведения глобальной оптимизации код имеет вид:

```
Выполнить N раз: { s=s+(*coef)*(*data); data++; coef++; }
```

Обозначим операции тела цикла как S_n^m, где n – номер операции, m – номер итерации, m может не обозначаться, если номер итерации может быть любым. Обозначим как операцию S_1: **coef*; S_2: **data*; S_3: (**coef*)*(**data*); S_4: *s*+(**coef*)*(**data*); S_5: *coef*++; S_6: *data*++. Этапы формирования программного конвейера проиллюстрируем с помощью таблицы, в каждой строке которой помещены операции, исполняющиеся параллельно. Строка, выделенная рамкой, обозначает границу. Выше границы формируется пролог цикла. Рассмотрим формирование конвейера для приведённого примера на рисунке ниже:

Таблица 6. Схема формирования программного конвейера алгоритмом EPS.

	Этап 1	Этап 2	Этап 3	Конвейер
Такт 1	$S_1^1\ S_5^1\ S_2^1\ S_6^1$	$S_1^1\ S_5^1\ S_2^1\ S_6^1$	$S_1^1\ S_5^1\ S_2^1\ S_6^1$	$S_1^1\ S_5^1\ S_2^1\ S_6^1$
Такт 2	S_3^1	$S_1^2\ S_5^2\ S_2^2\ S_6^2\ S_3^1$	$S_1^2\ S_5^2\ S_2^2\ S_6^2\ S_3^1$	$S_1^2\ S_5^2\ S_2^2\ S_6^2\ S_3^1$
Такт 3	S_4^1	S_4^1	$S_1^3\ S_5^3\ S_2^3\ S_6^3\ S_3^2\ S_4^1$	$S_1^3\ S_5^3\ S_2^3\ S_6^3\ S_3^2\ S_4^1$
Такт 4				$S_3^3\ S_4^2$
Такт 5				S_4^3

Справа в столбике «конвейер» показан конечный вид цикла, тело цикла затемнено. Выше тела сформирован пролог, ниже – эпилог. Интервал инициации итераций равен 1 (длина цикла в командах), время исполнения итерации составляет 3 такта, экономии времени составляет 200% от начального значения.

В случае, если конвейеризируется гнездо циклов, более вложенный цикл представляется как одна комплексная команда. Далее с учетом зависимостей между итерациями происходит конвейеризация внешнего цикла. Естественно, эта операция имеет смысл только в том случае, если архитектура процессора имеет достаточно длинное командное слово – необходим резерв скалярного параллелизма для конвейеризации тела цикла.

Кодогенератор (backend compiler)

Кодогенератор является одной из трёх основных частей компилятора, и последним принимает участие в генерации объектного кода. На модуле кодогенератора лежит ответственность за генерацию (суб)оптимального кода для данной процессорной архитектуры из оптимизированного модулем глобальной оптимизации внутреннего представления программы. Особенностью кодогенератора является его зависимость от архитектуры процессора и парадигмы генерации кода для него.

Структурно, кодогенератор может состоять из нескольких основных и достаточно большого количества дополнительных модулей, необходимых для поддержки кодогенерации для конкретного процессора.

Перечислим основные модули кодогенератора:

1. модуль выбора инструкций – сопоставляет операторам исходной программы инструкции физического процессора;
2. модуль определения класса регистров – определяет, в каком типе регистров должна обрабатываться переменная;
3. модуль распределения регистров – привязывает фактически обрабатываемые переменные к физическим регистрам процессора;
4. модуль генерации расписания команд – генерирует упорядоченную последовательность инструкций процессора для последующего выполнения.

В случае, если компилятор является перенацеливаемым – то есть, способен генерировать объектный код согласно имеющемуся описанию архитектуры – дополнительно имеется база данных, содержащая информацию об архитектуре процессора и его системе команд.

Перед рассмотрением методики генерации кода, рассмотрим кратко описание архитектуры микропроцессора в том виде, в котором оно используется в перенацеливаемых компиляторах.

Описание архитектуры микропроцессора

Микропроцессор описывается с помощью специального высокоуровневого языка описания архитектуры (Architecture Description Language) - ЯОА. Так как стандартов на ЯОА не существует, и каждый разработчик перенацеливаемого компилятора или системы совместной разработки аппаратного и программного обеспечения обычно имеет свой собственный ЯОА.

ЯОА делятся на 3 типа: структурные, бихеовиоральные и смешанные.

1. Структурные ЯОА: описание производится на структурном уровне в виде устройств (сумматор, и т.д.) и соединений между ними. Примеры языков: MIMOLA (компилятор MSSQ и RECORD), XASM (симулятор BUILDABONG);
2. Бихевиоральные ЯОА: описывается функционирование процессора. Обычно бихевиоральное описание состоит из описания ресурсов (регистров, памяти) и возможных преобразований содержимого этих ресурсов (фактически система инструкций процессора). Примеры языков: nML (IMEC, Cadence скорее для симуляторов, ассемблеров и дизассемблеров), ISDL (проект SPAM), FlexWare, LISA, Expression;
3. Смешанные ЯОА: имеющие черты как структурного, так и бихевиорального ЯОА. Примеры языков: PRMDL (Philips, архитектура TriMedia), HMDES.

Описание конвейера в GCC

Для повышения производительности современные микропроцессоры могут выполнять несколько различных инструкций одновременно, что достигается за счет использования нескольких функциональных устройств и конвейеризации исполнения в функциональных устройствах. Очевидно, что инструкция может быть запущена на исполнение, если выполнены два условия: входные данные для инструкции готовы к использованию и есть свободные функциональные устройства для ее исполнения. Следовательно, в процессе исполнения могут возникнуть два типа задержек: задержки по готовности данных (data delay) и задержки по занятости ресурсов (resource delay).

В оптимизирующих компиляторах специальный модуль – планировщик инструкций - отвечает за уменьшение задержек по занятости ресурсов и готовности данных. Эта достигается, в основном, за счет переупорядочивания инструкций, хотя могут быть использованы и другие методы. В состав планировщика инструкций входит важный компонент – распознаватель конфликтов в конвейере (pipeline hazard recognizer), отвечающий за определение возникающих ресурсных задержек.

Распознаватель конфликтов

В настоящее время даже в рамках одной архитектуры существует множество модификаций процессоров. Становится невозможным написание распознавателя конфликтов в конвейере для каждого из них. Ситуация еще более усложняется в перенацеливаемых компиляторах для множества архитектур и множества конкретных процессоров. Поэтому в современный компилятор интегрировано описание модели конвейера целевого процессора и, как правило, генератор распознавателя конфликтов.

Сначала в GCC распознаватель конфликтов управлялся таблицами занятости функциональных устройств, сгенерированными с помощью файлов описания процессора. Таблицы были самым простым методом описания процессора, но значительно огрубляли описание, и чем более сложным становилось описание процессора, тем более медленным становился распознаватель конфликтов, основанный на использовании таблиц. С дальнейшим усложнением описания процессора скорость работы распознавателя конфликтов стала существенной проблемой.

Модель распознавателя конфликтов и описание конвейера

Модель конвейера основана на использовании описания всех комбинаций резервирования функциональных устройств (ФУ), использующихся инструкциями, с помощью таблиц резервирования. Таблица резервирования фактически представляет собой последовательность занятий функциональных устройств в дискретные моменты времени, прошедшие со времени начала исполнения инструкции в процессоре – фактически описывается, в какой такт от начала инструкции занимается некоторое функциональное устройство. Длина таблицы для инструкции не превышает времени выполнения инструкции. Пример таблицы резервирования приведен ниже.

Реализация распознавателя конфликтов основывается на использовании конечных автоматов, построенных на базе таблиц резервирования. Каждое состояние автомата отражает существующее резервирование функциональных устройств и все возможности бесконфликтного исполнения инструкций в этом состоянии в виде переходов в иные состояния автомата. Если два состояния соединены дугой (маркировка дуги соответствует выполняемой инструкции), то эта инструкция может быть выполнена в текущем состоянии автомата и её выполнение не вызовет конфликтов по ресурсам с инструкциями, которые в данный момент выполняются в конвейере. Из каждого состояния автомата

проведена дуга, маркированная *'следующий такт'* – она используется в случае, если нет возможности в этом такте выполнить какую-либо инструкцию.

Рассмотрим гипотетический процессор, состоящий из трёх устройств: целочисленного арифметико-логического устройства (АЛУ), АЛУ для чисел с плавающей запятой и функционального устройства (ФУ) доступа к памяти (Рисунок 17).

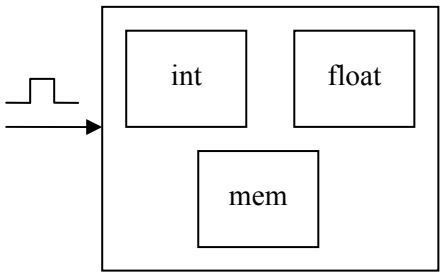


Рисунок 17. Структурная схема гипотетического процессора

Процессор может выполнять следующие инструкции:

a) целочисленная арифметико-логическая операция занимает на 1 такт целочисленное АЛУ

b) арифметико-логическая с плавающей точкой занимает на 1 такт АЛУ для чисел с плавающей запятой

c) загрузка значения из памяти или сохранение в памяти в течение первого такта занимает целочисленное АЛУ и устройство доступа к памяти, в течение второго такта только устройство доступа к памяти. Соответствующая схема резервирования представлена в таблице 7.

Таблица 7. Таблица резервирования для гипотетического процессора.

Класс инструкции	**Занятые устройства**	
	1 такт	**2 такт**
Целочисленная (i)	int	-
с плавающей запятой (f)	float	-
Загрузка из памяти (ls)	mem + int	mem

Конечный автомат для описанного процессора представлен на Рисунке 18 (0 – означает, что устройство “доступно”, x – “не доступно”). Подробности о составление автомата на базе таблиц резервирования представлены в [24].

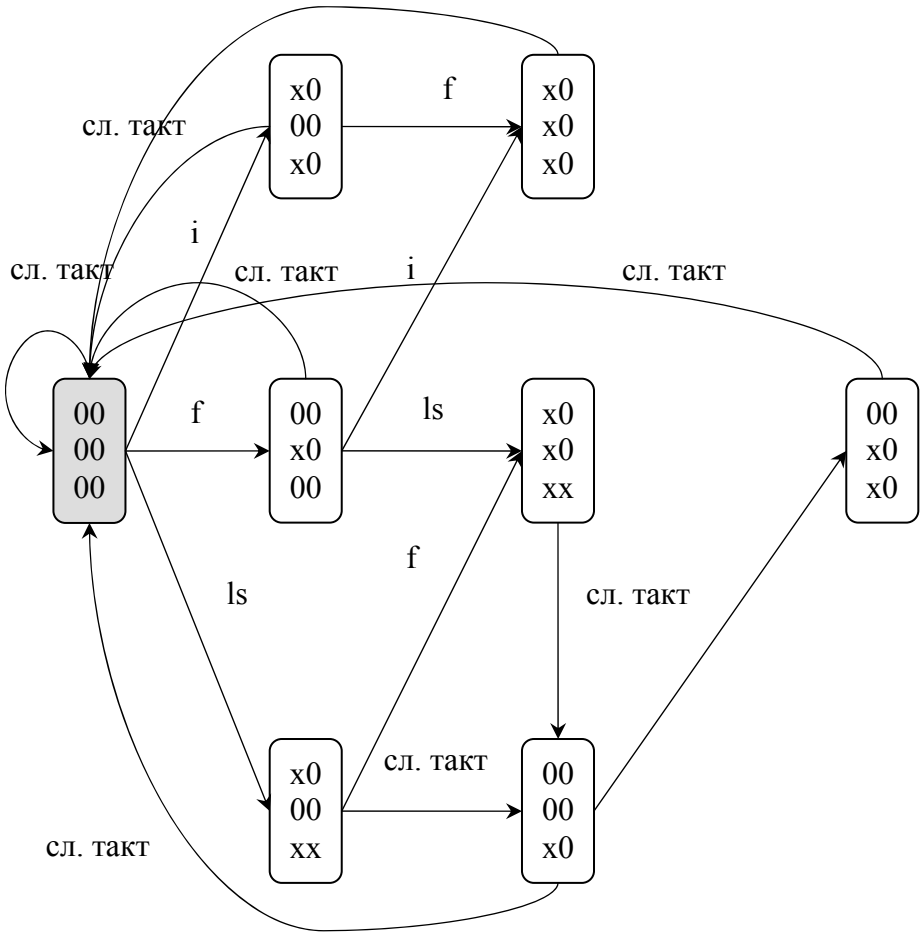


Рисунок 18. Автомат для определения конфликтов при исполнении инструкций

Для того чтобы определить ресурсные задержки планировщику инструкций достаточно пройти из начального состояния по дугам, соответствующим инструкциям или дугам, маркированным ‘сл. такт’, если невозможно выполнить инструкцию. Фактически формирователь расписания инструкций прямо использует автомат при генерации расписания инструкций.

Модель описания конвейера в GCC

В GCC описание процессора производится с помощью Lisp-подобного языка. Синтаксис основных конструкций, необходимых для описания процессора, приведен в Таблице 8.

Таблица 8. Синтаксис описания модели конвейера.

Конструкция	Описание конструкции	Параметры конструкции
(define_automaton AUTOMATON-NAME)	определение автомата — распознавателя конфликтов	AUTOMATON-NAME – строка, название автомата
(define_cpu_unit UNIT-NAMES AUTOMATON-NAME)	определение функциональных устройств процессора	UNIT-NAMES – строка, название функционального устройства AUTOMATON-NAME – название автомата, куда помещается описываемое устройство
(define_insn_reservation INSN-NAME DEFAULT-LATENCY CONDITION REGEXP)	определение резервирования функциональных устройств для инструкций	DEFAULT-LATENCY – число, определяющее латентность (время исполнения) инструкции INSN-NAME – строка, внутреннее имя инструкции CONDITION – определяет какие RTL инструкции описываются (класс инструкций) REGEXP – описывает, какие функциональные устройства процессора будут зарезервированы инструкцией
(define_reservation RESERVATION-NAME REGEXP)	определение резервирования функциональных устройств классом инструкций	RESERVATION-NAME – строка, имя резервирования REGEXP – описывает, какие функциональные устройства процессора будут зарезервированы инструкцией

(exclusion_set UNIT-NAMES UNIT-NAMES) (presence_set UNIT-NAMES PATTERNS) (absence_set UNIT-NAMES PATTERNS)	определение дополнительных ограничений, налагаемых на ресурсы при исполнении инструкций	UNIT-NAMES – строка, перечисление функциональных устройств PATTERNS – строка, шаблоны функциональных устройств, перечисленных через запятую. Шаблон – одно устройство или группа устройств, перечисленных через пробел

При описании конвейера сначала определяется автомат с помощью `define_automaton`. Файл описания может содержать несколько автоматов, но все они должны иметь уникальные имена. На практике различные автоматы применяются для описания различных процессоров одной архитектуры. Иногда несколько автоматов применяются для описания одного процессора.

Таблица 9. Синтаксис описания резервирования устройств.

regexp = regexp “,” oneof | one of	Символ “,” используется для разделения резервирования по тактам
alloff = allof “+” repeat | repeat	Символ “+” используется для описания того, что резервирование определяется и первым регулярным выражением, и вторым регулярным выражением и т.д.
oneof = oneof “|” allof | allof	Символ “|” используется для описания того, что резервирование определяется или первым регулярным выражением или вторым регулярным выражением и т.д.
repeat = element “*” number | element	Символ “*” – указывает на то, что element будет зарезервирован number тактов
element = cpu_unit_name | reservation_name | result_name | “nothing” | “(“ regexp “)”	“nothing” – функциональные устройства не резервируются cpu_unit_name – резервируется соответствующее функциональное устройство

Для описания характеристик занятия конвейера классом инструкций применяется конструкция `define_insn_reservation`. Для описания резервирования используются регулярные выражения, в Таблице 9 приведен их синтаксис.

Рассмотрим описание конвейера на примере гипотетического процессора.

Пример описания

В качестве примера рассмотрим фрагмент описания суперскалярного RISC-процессора, который может выбирать на исполнение три инструкции (две целочисленных и одну с плавающей запятой) за один такт, но может завершить выполнение только двух инструкций. Схематически такой процессор представлен на Рисунке 19. Фрагмент описания конвейера приведен ниже.

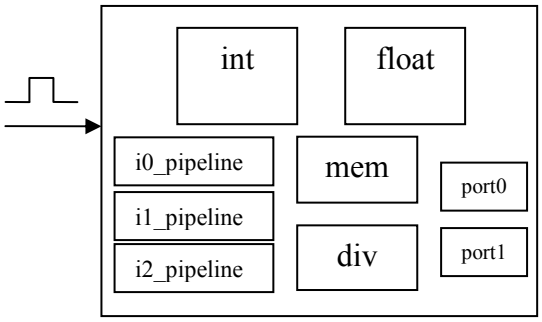


Рисунок 19. Структурная схема гипотетического процессора.

Рассмотрим описания функциональных устройств и инструкций.

Порты декодирования инструкций:

```
1: (define_cpu_unit “i0_pipeline, i1_pipeline”)
```

Целочисленное АЛУ не описывается.

АЛУ с плавающей запятой и выходные порты:

```
2: (define_cpu_unit “f_pipeline, port0, port1”)
```

Устройство “деление”

```
3: (define_cpu_unit “div”)
```

Простая целочисленная инструкция, исполняющаяся 2 такта:

```
4: (define_insn_reservation “simple” 2
```

```
        (eq_attr "cpu" "int")
        "(i0_pipeline|i1_pipeline),(port0|port1)")
```

Инструкция умножения (полностью конвейеризирована, потому не используется выделенное устройство-умножитель):

```
5: (define_insn_reservation "mult" 4
        (eq_attr "cpu" "mult")
        "i1_pipeline, nothing*2,(port0|port1)")
```

Инструкция деления, (не конвейеризирована):

```
6: (define_insn_reservation "div" 9
        (eq_attr "cpu" "div")
        "i1_pipeline, div * 7, div + (port0|port1)")
```

Инструкции обработки чисел с плавающей запятой:

```
7: (define_insn_reservation "float" 3
        (eq_attr "cpu" "float")
        "f_pipeline, nothing, (port0|port1)")
```

Все простые целочисленные инструкции могут запускаться через декодер в целочисленных конвейерах, результат выполнения будет доступен через два такта (определение 4). Сначала целочисленная инструкция пытается занять функциональное устройство i0_pipeline, если оно занято делается попытка занять функциональное устройство i1_pipeline. Целочисленное умножение и деление могут исполняться только во втором целочисленном конвейере, их результат выполнения будет получен через 4 и 9 тактов соответственно (определения 5 и 6). Инструкция умножения (определение 5) полностью конвейеризирована – т.е. новая инструкция умножения может быть подана на исполнение в каждом такте. Целочисленное деление не конвейеризировано (т.е. невозможно запустить инструкцию деления, пока не отработала предыдущая аналогичная инструкция). Операции с плавающей запятой полностью конвейеризированы, результат их доступен через 3 такта (определение 7).

Описание целевой машины в GCC на примере микроконтроллера семейства AVR.

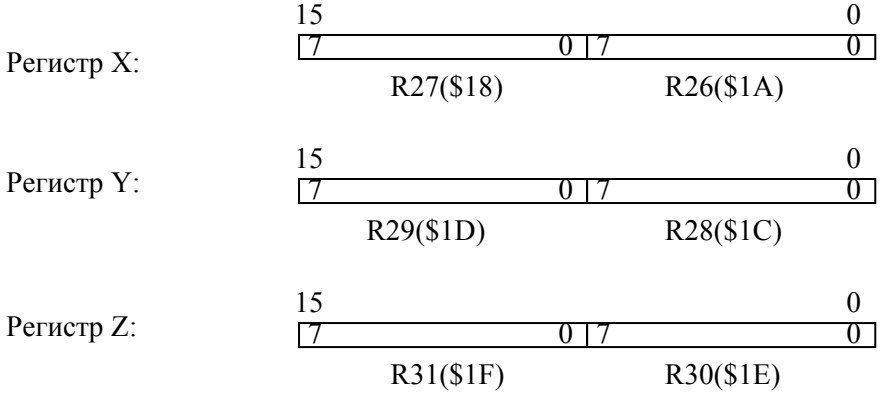
AVR – это новое семейство 8-разрядных RISC-микроконтроллеров фирмы Atmel. Они отличаются сравнительно большой скоростью работы и большей универсальностью. Очень быстрая гарвардская RISC-архитектура загрузки и выполнения большинства инструкций в течение одного цикла тактового генератора. Смысл её состоит в том, что память программ и данных располагается в разных областях памяти. Так во время выполнения одной команды следующая выбирается из памяти. Отсутствует внутреннее деление частоты (если использован кварцевый резонатор 16 МГц, то быстродействие будет почти 16 MIPS). Программы содержатся в электрически перепрограммируемой постоянной памяти FLASH ROM. Система команд микроконтроллеров AVR изначально проектировалась с учетом языка программирования высокого уровня C. Микроконтроллер имеет 32 регистра, каждый из которых напрямую работает с АЛУ. Имеются относительные команды переходов и ветвлений, что позволяет получать перемещаемый код. Отсутствует необходимость переключать страницы памяти.

Кроме регистровых операций, для работы с регистровым файлом могут использоваться доступные режимы адресации, так как регистровый файл занимает адреса $00-$1F в области данных, обращаться к ним можно и как к ячейкам памяти.

Пространство ввода/вывода состоит из 64 адресов для периферийных функций процессора, таких, как управляющие регистры, таймеры/счетчики и др. Доступ к пространству ввода/вывода может осуществляться непосредственно как к ячейкам памяти, расположенным после регистрового файла ($20-$5F).

Большинство команд, использующих регистры, могут использовать любые регистры общего назначения. Исключение составляют пять команд оперирующих с константами: SBCI, SUBI, CPI, ANDI, ORI и команда LDI. Эти команды работают только со второй половиной регистрового файла – R16…R31.

Шесть из 32 регистров – R26…R31 – можно использовать как три 16-разрядных адресных указателя в адресном пространстве данных. Эти регистры обозначаются как X, Y, Z. Регистр Z можно использовать для адресации таблиц в памяти программы.



Эти регистры могут использоваться как фиксированный адрес для адресации с автоинкрементном или с автодекрементном.

Большая часть команд имеет размер 16 разрядов. Каждый адрес в памяти программы содержит одну 16 или 32-разрядную команду.

При обработке прерываний и вызове подпрограмм адрес возврата запоминается в стеке, размещаемом в оперативный памяти SRAM, или в реализуемом аппаратно стеке глубиной 3, для микроконтроллеров без SRAM.

Минимальное время реакции на любое из предусмотренных в процессоре прерываний – 4 такта.

Детальную информацию о микроконтроллерах AVR можно получить на сайте производителя www.atmel.com.

Теперь перейдём к тому, как рассмотренная архитектура AVR представляется в GCC.

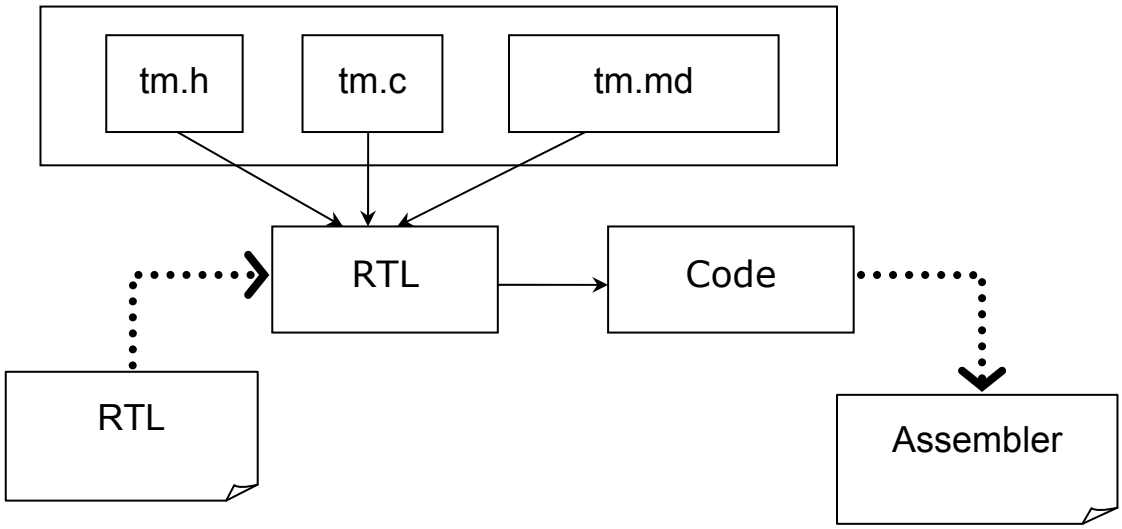


Рисунок 20

В GCC описание процессора (класса процессоров) выделено в несколько отдельных от основной реализации back end'а файлов, а именно: `tm.h`, `tm.c`, `tm.md` (см. рисунок 20). Все они находятся в каталоге `gcc/config/tm`, где `tm` – это мнемоническое имя для целевого процессора или семейства процессоров, например `i386`, `alpha`, `sparc`, `vax`. Описываемое нами семейство AVR находится в одноимённом каталоге `avr`. Таким образом, в этом разделе мы рассмотрим некоторые примеры описаний из файлов `avr.h`, `avr.c`, `avr.md`.

Файл tm.h

Описание процессора и ABI (Application Binary Interface) производится с помощью макросов в файле ***`tm.h`***. Этими макросами описываются категории, которые перечислены ниже.

- *Окружение компилятора*. На пример, синтаксис ассемблера, каталог, где искать заголовочные файлы и системные библиотеки.
- *Основные свойства машины (Fundamental machine properties)*, такие как, порядок байт (прямой или обратный), адресуемое пространство памяти, количество и тип регистров, а так же режимы адресации.
- *ABI*. Описываются способы вызова функций.

Поддержка описания машины – дополнительные средства, которые непосредственно используются в md-файле или при работе с внутренним представлением.

Теперь приведём несколько примеров из перечисленных категорий. Ниже рассмотрены лишь некоторые макросы; полное описание можно найти в файле ***avr.h***.

`FIXED_REGISTERS`

Макрос описывает регистры, за которыми закреплена определённая функциональность, т.е. такие регистры не доступны для обычного использования. Например, типичными регистрами такого рода являются: указатель на стек (stack pointer), указатель на кадр стека, счетчик команд и т.п.

Описание представляет собой последовательность чисел, разделенных запятыми, заключенная в фигурные скобки (это инициализатор для обычного массива языка C). Каждый элемент относится к соответствующему регистру. Если этот элемент равен 1, то соответствующий регистр специальный, иначе – общего назначения.

```
#define FIXED_REGISTERS {         \
  1,1,  /* r0 r1 */               \
  0,0,  /* r2 r3 */               \
  0,0,  /* r4 r5 */               \
  0,0,  /* r6 r7 */               \
```

```
  0,0,  /* r8 r9 */                \
  0,0,  /* r10 r11 */              \
  0,0,  /* r12 r13 */              \
  0,0,  /* r14 r15 */              \
  0,0,  /* r16 r17 */              \
  0,0,  /* r18 r19 */              \
  0,0,  /* r20 r21 */              \
  0,0,  /* r22 r23 */              \
  0,0,  /* r24 r25 */              \
  0,0,  /* r26 r27 */              \
  0,0,  /* r28 r29 */              \
  0,0,  /* r30 r31 */              \
  1,1,  /* вершина и кадр стека */ \
  1,1   /* указатели на аргументы */  }
```

В этом примере объявлены следующие фиксированные регистры: `r0, r1, указатели стека, указатели на аргументы`.

CALL_USED_REGISTERS

Указывает регистры, которые могут терять своё значение при вызове функций. Другими словами, указываются регистры, значения которых нужно сохранить при вызове функции. Описание аналогично предыдущему. Если для регистра соответствующий элемент равен 0, то компилятор автоматически сохранит его значение в теле функции, и восстановит его значение перед выходом из неё, если значение регистра изменилось.

```
#define CALL_USED_REGISTERS {                  \
    1,1,  /* r0 r1 */                          \
    0,0,  /* r2 r3 */                          \
    0,0,  /* r4 r5 */                          \
    0,0,  /* r6 r7 */                          \
    0,0,  /* r8 r9 */                          \
    0,0,  /* r10 r11 */                        \
    0,0,  /* r12 r13 */                        \
    0,0,  /* r14 r15 */                        \
    0,0,  /* r16 r17 */                        \
    1,1,  /* r18 r19 */                        \
    1,1,  /* r20 r21 */                        \
```

```
    1,1,  /* r22 r23 */                          \
    1,1,  /* r24 r25 */                          \
    1,1,  /* r26 r27 */                          \
    0,0,  /* r28 r29 */                          \
    1,1,  /* r30 r31 */                          \
    1,1,  /*  STACK */                           \
    1,1   /* arg pointer */  }
```

Регистры с `r18` по `r27` и `r30`, `r31` не будут сохранены при вызовах функций.

`REG_CLASS_CONTENTS`

Этот макрос описывает классы регистров. Для каждого класса указываются регистры входящие в него. Описание, так же как и в предыдущих случаях, представляет собой инициализатор обычного C-массива: каждый элемент соответствует классу регистров.

Принадлежность регистра с номером R классу K, определяется из истинности условия `REG_CLASS_CONTENTS[K] & (1 << R) == 1`,

где `REG_CLASS_CONTENTS[K]` – элемент описания с номером K, который представляется как битовая маска.

```
1: #define REG_X 26
2: #define REG_Y 28
3: #define REG_Z 30
4: #define REG_W 24

5: #define REG_CLASS_CONTENTS {                                          \
6:  {0x00000000,0x00000000},      /* NO_REGS */                           \
7:  {0x00000001,0x00000000},      /* R0_REG */                            \
8:  {3 << REG_X,0x00000000},      /* POINTER_X_REGS, r26 - r27 */         \
9:  {3 << REG_Y,0x00000000},      /* POINTER_Y_REGS, r28 - r29 */         \
10: {3 << REG_Z,0x00000000},      /* POINTER_Z_REGS, r30 - r31 */         \
11: {0x00000000,0x00000003},      /* STACK_REG, STACK */                  \
12: {(3 << REG_Y) | (3 << REG_Z),                                         \
        0x00000000},               /* BASE_POINTER_REGS, r28 - r31 */     \
13: {(3 << REG_X) | (3 << REG_Y) | (3 << REG_Z),                          \
        0x00000000},               /* POINTER_REGS, r26 - r31 */          \
14: {(3 << REG_X) | (3 << REG_Y) | (3 << REG_Z) | (3 << REG_W),           \
```

```
        0x00000000},                /* ADDW_REGS, r24 - r31 */              \
15: {0x00ff0000,0x00000000},        /* SIMPLE_LD_REGS r16 - r23 */          \
16: {(3 << REG_X)|(3 << REG_Y)|(3 << REG_Z)|(3 << REG_W)|(0xff << 16),      \
        0x00000000},        /* LD_REGS, r16 - r31 */                        \
17: {0x0000ffff,0x00000000},        /* NO_LD_REGS  r0 - r15 */              \
18: {0xffffffff,0x00000000},        /* GENERAL_REGS, r0 - r31 */            \
19: {0xffffffff,0x00000003}         /* ALL_REGS */                          \
20:}
```

В примере определяются следующие классы регистров: пустой класс (6), класс, состоящий из 0-го регистра (7), класс, состоящий из 26 и 27 регистров (8), и т.д. Заметим, что у AVR более 32 регистров, поэтому каждый из элементов в примере состоит не из одного числа-маски, а из двух.

INDEX_REG_CLASS

Макрос определяет имя класса, которому принадлежат все индексные регистры. Индексный регистр – это регистр, использующийся в адресных выражениях. В этих выражениях происходит его умножение на число и сложение с базовым адресом памяти.

```
#define INDEX_REG_CLASS NO_REGS
```

Пример показывает, что в AVR нет индексных регистров.

REGNO_OK_FOR_BASE_P

Этот макрос принимает в качестве своего аргумента номер регистра. Если регистр может использоваться как базовый регистр при адресации операндов, то возвращается ненулевое значение. Аргумент может быть как аппаратным регистром, так и псевдорегистром.

```
#define REGNO_OK_FOR_BASE_P(r)  (((r) < FIRST_PSEUDO_REGISTER           \
                                 && ((r) == REG_X                       \
                                     || (r) == REG_Y                    \
                                     || (r) == REG_Z                    \
                                     || (r) == ARG_POINTER_REGNUM))     \
                                || (reg_renumber                        \
                                    && (reg_renumber[r] == REG_X        \
```

```
|| reg_renumber[r] == REG_Y      \
|| reg_renumber[r] == REG_Z      \
|| (reg_renumber[r]              \
    == ARG_POINTER_REGNUM))))
```

В примере показано, что в качестве базового регистра могут использоваться аппаратные регистры: REG_X, REG_Y, REG_Z или ARG_POINTER_REGNUM и псевдорегистры, которые были распределены как эти аппаратные регистры.

Определение типов данных

Для определения длин типов данных используются несколько макросов. Размеры типов указываются в битах.

```
#define INT_TYPE_SIZE (TARGET_INT8 ? 8 : 16)
#define SHORT_TYPE_SIZE (INT_TYPE_SIZE == 8 ? INT_TYPE_SIZE : 16)
#define LONG_TYPE_SIZE (INT_TYPE_SIZE == 8 ? 16 : 32)
#define MAX_LONG_TYPE_SIZE 32
#define LONG_LONG_TYPE_SIZE 64
#define FLOAT_TYPE_SIZE 32
#define DOUBLE_TYPE_SIZE 32
#define LONG_DOUBLE_TYPE_SIZE 32
```

В примере устанавливаются длины основных типов данных.

Файл tm.c

Многие макросы имеют достаточно сложную реализацию, которая состоит не из одной инструкции и при его подстановке получается длинная строка кода. Реализация таких макросов походит на реализацию нетривиальной функции, код которой не имеет смысла вставлять в место использования макроса. Вместо того, что бы записывать весь код такого макроса в виде его непосредственной подстановки, делают функцию, в вызов которой и разворачивается данный макрос. Это упрощает отладку и уменьшает влияние контекста использования макроса на реализацию его функциональности. Функция имеет то же имя, что и макрос, но написанное строчными буквами.

В файле tm.c содержатся реализации таких функций, а также функции, которые служат поддержкой описаний в tm.md.

Рассмотрим описание макроса такого рода.

REG_CLASS_FROM_LETTER

Возвращает класс регистра по его символьному обозначению. В avr.h этот макрос реализован следующим образом:

```
#define REG_CLASS_FROM_LETTER(C) avr_reg_class_from_letter(C)
```

а в файле avr.c представлена реализация функции

avr_reg_class_from_letter:

```
enum reg_class
avr_reg_class_from_letter  (c)
  int c;
{
  switch (c)
   {
   case 't' : return R0_REG;
   case 'b' : return BASE_POINTER_REGS;
   case 'e' : return POINTER_REGS;
   case 'w' : return ADDW_REGS;
   case 'd' : return LD_REGS;
   case 'l' : return NO_LD_REGS;
   case 'a' : return SIMPLE_LD_REGS;
   case 'x' : return POINTER_X_REGS;
   case 'y' : return POINTER_Y_REGS;
   case 'z' : return POINTER_Z_REGS;
   case 'q' : return STACK_REG;
   default: break;
   }
  return NO_REGS;
}
```

Файл tm.md

Этот файл содержит описание машины, а именно:

- описание инструкций;
- описание атрибутов, которые используются в других описаниях;

- описание способов разделения сложных инструкций на последовательность простых;
- описание peephole-оптимизаций.

Все описания выполнены в формате RTL. Здесь мы рассмотрим лишь описание инструкций.

Синтаксис описания:

```
(define-тип-описания "(необязательно) имя-описания"
 [(set (результат-инструкции)
       (операнд))
  необязательные дополнительные set-выражения]
"необязательное условие-применимости-описания"
"шаблон выхода"
(необязательно: [атрибуты]))
```

`имя-описания` – имя шаблона. Имена должны быть уникальны. Возможно наличие анонимных шаблонов, т.е. без имени.

`тип-описания` – тип шаблона, различают два типа:

- `define_insn` – определение оператора, прямо соответствующего инструкции процессора;
- `define_expand` – определение оператора, которому соответствует несколько инструкций процессора.

`операнд` – любая комбинация операций. Такая комбинация, очевидно, является суперпозицией и имеет вид дерева. Листьями этого дерева являются непосредственные операнды. Здесь мы опишем лишь один часто используемый формат для этих операндов, а именно:

```
(match_operand : M номер-операнда "предикат" "ограничение")
```

где,

- `M` – режим (см. главу RTL);

- "предикат" – указывает на класс операндов, которые можно применять; перечислим несколько предопределённый классов:
  - register_operand – регистровый операнд;
  - address_operand – операнд, являющийся адресом;
  - immediate_operand – константный операнд (может быть константным адресом);
  - const_int_operand – целочисленный константный операнд;
  - const_double_operand – константный операнд, являющийся числом с плавающей точкой;
  - nonimmediate_operand – операнды, которые не входят в класс immediate_operand;
  - memory_operand – операнд в памяти;
  - general_operand – любой допустимый операнд: регистр, константа, память;
- "ограничение" – указывает разрешённые комбинации операндов и дополнительные ограничения. Представляет собой строку символов, возможно с запятыми, которые отделяют различные варианты (альтернативы) использования данного операнда. Перечислим некоторые элементы ограничений (символы):
  - '0' ... '9' – этот операнд должен быть тем же, что и операнд с указанным номером;
  - 'r' – это регистр из GENERAL_REGS;
  - 'm' – это операнд из memory_operand;

- `'='` – признак того, что операнд может потерять своё старое значение; имеет силу над всеми альтернативами;
- `'%'` – этот операнд может быть переставлен местами со следующим операндом; например, это имеет место для коммутативных операций, таких как сложение, побитовое ИЛИ и т.п.

`результат-инструкции` – выход (результат) инструкции

`дополнительные set-выражения` – дополнительные выражения вида `(set …)`; предполагается, что они будут выполняться параллельно, поэтому выход одного из них нельзя использовать на входе другого.

`условие-применимости-описания` – условие срабатывания шаблона; при его истинности этот шаблон можно применять.

`шаблон выхода` – целевая последовательность ассемблерных инструкций или код на C, который генерирует эту последовательность.

`атрибуты` – атрибуты инструкции; устанавливаются значения атрибутов, если значения по умолчанию не подходят.

Теперь рассмотрим несколько примеров описаний из AVR.

```
(define_insn "negqi2"
  [(set (match_operand:QI 0 "register_operand" "=r")
        (neg:QI (match_operand:QI 1 "register_operand" "0")))]
  ""
  "neg %0"
  [(set_attr "length" "1")
(set_attr "cc" "set_zn")])
```

В данном примере описывается инструкция `negqi2`, которая обращает знак операнда. Операнд имеет размер байта (`QI`), должен находиться в регистре (`"register_operand"`). Далее видно, что приёмник результата должен совпадать с источником значения (у операнда-источника имеется ограничение `'0'`, которое указывает на то, что это тот же регистр, что и стоящий на первом месте, т.е. регистр-результат), и, что естественно, это значение может измениться в результате выполнения операции (на что указывает `'='`).

Оптимизации в компиляторе

Определение оптимизирующего преобразования

Преобразованием программы является любая функция *p*, определённым образом изменяющая внутреннее представление программы. Существует общий класс конкретизирующих преобразований (обобщающих, сужающих и эквивалентных), использующийся как при доказательстве правильности программ, так и при оптимизирующих трансформациях. Подклассом конкретизирующих преобразований, является класс *анализирующих* преобразований, направленных на решение задач верификации и на решение задач потокового анализа при оптимизации программ.

На практике из класса конкретизирующих преобразований используются некоторые основные подклассы, например:

- оптимизирующие преобразование, направленные на повышение эффективности программ (улучшающие программу в традиционном смысле – уменьшением времени исполнения и размера генерируемого кода – с учётом разнообразия платформ и сред исполнения);
- преобразования, назначением которых является повышение самодокументируемости и наглядности аннотируемой программы (пополнение программы утверждениями о её свойствах и преобразование базисной программы с помощью переименования объектов, вставок явных описаний, улучшения структуры программы);

- преобразования, осуществляющие построение отладочной версии программы (с пополнением базисной программы набором операторов, проверяющих справедливость свойств программы, указанных в аннотациях, пример аннотации – макрос *assert*.).

Оптимизирующее преобразование в компиляторе исполняется в три этапа:

1. определить часть программы, которую надо оптимизировать, и определить соответствующее оптимизирующее преобразование;
2. проверить, что преобразование не изменяет результат исполнения данного участка кода или изменяет его в рамках, утверждённых пользователем;
3. провести преобразование.

Пункт 1 является предметом длительных исследований, так как его исполнение зависит от множества факторов (в т.ч. архитектура процессора). Пункт 2 гласит о том, что результат работы программы не должен изменяться. Наиболее слабое определение гласит что:

Преобразование допустимо, если оригинальная и преобразованная программы дают один и тот же вывод для разных исполнений при одинаковых входных данных.

Уточнение 1.1. Два запуска программы являются идентичными, если они произведены при одинаковых входных данных и если каждая пара соответствующих операторов с недетерминированным исполнением при обоих запусках даёт одинаковый результат.

Обычно недетерминированные исполнение является результатом вызова внешних функций, например процедур операционной системы.

Рассмотрим некоторые возможные нарушения корректности выполнения программы при преобразованиях:

Переполнение.

`for (i=0; i<n; i++)`
`{`
`  a[i] = a[i] + b[k] + 2.5;`
`}`
	`temp = b[k] + 2.5;`
	`for (i=0; i<n; i++)`
	`{`
	`  a[i] = a[i] + temp;`
	`}`

Недостоверно, что `a[i]+(b[k]+2.5)`, вместо версии `(a[i]+b[k])+2.5` всегда будет исполнено без переполнения. Возможно, суммирование `(b[k]+2.5)` даст переполнение сразу же, в отличие от непреобразованной версии, и результат исполнения программы будет отличаться.

Разные результаты могут получаться из-за изменения порядка следования операций в программе, и ошибка может накапливаться. Здесь же необходимо вспомнить про ошибки округления.

Ошибка при работе с памятью. В предыдущем примере в первом случае, если индекс *k* выходит за допустимые пределы, но *n*=0 цикл выполняться не будет, и адресации памяти по неверному адресу не будет. Как только мы выносим этот инвариант за пределы цикла, имеем ошибку.

Различные результаты в случае, если массивы *a* и *b* перекрываются.

Примеры некорректных преобразований можно перечислять бесконечно, поэтому на практике применяют несколько другое правило:

Преобразование программы корректно, если для всех семантически правильных исполнений программы, оригинальная и преобразованная версии программы дают идентичный результат для идентичных запусков.

Фактически, при программировании мы делаем множество допущений, которые могут в конкретном случае и не выполняться: например, индексация массива может не выходить за пределы его размерностей. Корректность в этом контексте является не свойством самой программы, а

свойством её конкретного запуска – поскольку при одних входных данных программа может вести себя корректно, при других некорректно. Те же самые соображения относятся и к обработке исключений. Вообще недостоверно, что преобразованная программа будет давать абсолютно такой же результат, как и исходная – возможно это не требуется.

Таким образом, на практике используется ещё более слабое утверждение:

Преобразование программы корректно, если для всех семантически корректных запусков исходной программы, оригинальная и преобразованная версии производят эквивалентные операции при идентичных запусках. Все перестановки коммутативных операций подразумеваются эквивалентными.

Естественно, если в случае выполнения компилятором пункта (3) появляется погрешность в вычислениях, выходящая за определённые рамки, необходимо вернуться к пункту (2) – трудно заранее предугадать, как будет оптимизировано исполнение коммутативных операторов.

Участки экономии.

Оптимизирующие преобразования обычно производятся над некоторым участком программы, и, в зависимости от типа преобразования, рассматриваются участки различной структуры и величины. Основные градации величины участков экономии такие:

1. оператор – в основном арифметические операторы являются участком экономии в рамках одного оператора;
2. базовый блок – последовательность операторов с единственной точкой входа и без ветвлений, базовый блок (ББ) был объектом многих ранних исследований по оптимизации;

3. самый вложенный цикл, в этом контексте выполняются многие оптимизации;
4. идеально вложенное гнездо циклов (все циклы, кроме самого вложенного содержат в себе только один оператор – более глубоко вложенный цикл);
5. вложенное гнездо циклов (любого формата);
6. процедура (глобальная оптимизация);
7. множество процедур, рассматриваемых вместе (межпроцедурная оптимизация).

Каждое оптимизирующее преобразование применяется с целью улучшения некоторой характеристики (характеристик) программы. Зачастую преобразование при улучшении одной из характеристик, ухудшает другую. Приведём примеры характеристик:

1. количество занимаемых ресурсов процессора (например, функциональных устройств);
2. минимизация количества выполняемых операций;
3. минимизация количества обращений в оперативную память;
4. минимизация размера программы и использования памяти для данных;
5. минимизация энергопотребления.

Базовым понятием, используемым во всех оптимизирующих преобразованиях уровня базового блока и выше, является понятие зависимости по данным.

В общем случае встречаются три типа зависимости по данным: истинная или потоковая, антизависимость, и зависимость по выходу (выходная). Для каждого типа зависимости определены исток (источник зависимости) и

сток зависимости (зависящий оператор). *Графом зависимости по данным* является ориентированный граф, вершинами которого являются операторы программы, две вершины S_i и S_j соединены дугой, если существует зависимость одного из трёх перечисленных типов с истоком в S_i и стоком в S_j. Каждая дуга имеет метку, определяющую тип зависимости и глубину зависимости (для оператора в теле гнезда цикла) – номер цикла, порождающего данную зависимость.

Пусть S_i обозначает i-й оператор в программе или ББ, если считать в лексикографическом порядке. Все имеющиеся в программе операторы разделяются на два типа: *скалярные* и *индексированные*. Индексированные операторы – те, которые встречаются в теле какого-нибудь цикла или их исполнение управляется с помощью индексной переменной (векторные операторы). Остальные операторы являются скалярными. *Степенью* оператора является число разных циклов, его окружающих, или число измерений его операндов. Оператор степени k имеет вид $S_i(I_1,I_2,\ldots,I_k)$, где I_j, $L_j \le I_j \le U_j$, – индексная переменная j-го цикла, L_j и U_j – нижняя и верхняя границы изменения индексной переменной. Индексированный оператор $S_i(I_1,I_2,\ldots,I_k)$ имеет $\prod_{j=1}^{k} N_j$, где $N_j = U_j - L_j + 1$, разных экземпляров (исполнения) по одному для каждого значения I_j, $j=1,\ldots,k$. Первый и последний экземпляры оператора степени k имеют вид $S_i(L_1,L_2,\ldots,L_k)$ и $S_i(U_1,U_2,\ldots,U_k)$.

Порядок исполнения между двумя операторами S_i и S_j определяется следующим образом: скалярный оператор S_i исполняется раньше скалярного S_j (обозначается $S_i \preceq S_j$), если оператор S_i лексически предшествует S_j ($i<j$). Для индексированных операторов степени k имеет место:

если S_i и S_j не имеют общих индексов, то $S_i \preceq S_j$, если $i \le j$;

если у них одни и те же индексы, то (обозначая как $S_i(i_1,...,i_k)$ конкретный экземпляр оператора S_i при $I_1=i_1,...,I_k=i_k$: $S_i(i_1,...,i_k) \preceq S_j(j_1,...,j_k)$ если $i \leq j$ и существует такое m ($0 \leq m \leq k$), что $i_l=j_l$ для $l=1,...,m$ и $i_m+1<j_m+1$. Если же $i<j$, всё остаётся аналогично, но с условием $m<k$;

если S_i и S_j имеют $m<k$ общих индексов и $i_l=j_l$ ($l=1,...,m$), то $S_i \underline{\prec} S_j$ только тогда, когда $i \leq j$.

Определим также множества $IN(S)$ и $OUT(S)$ входных и выходных переменных оператора S_i соответственно. Определим через $OUT(S_i(i_1,...,i_k))$ множество экземпляров переменных (необязательно разных), определяющихся экземпляром $S_i(i_1,...,i_k)$ оператора S_i. Аналогично, определим через $IN(S_i(i_1,...,i_k))$ множество экземпляров переменных, которые используются тем же экземпляром оператора. Считаем, что оператор S_i простой (содержит не более одного присваивания) если $||OUT(S_i)|| \leq 1$.

Два оператора $S_i(I_1,I_2,...,I_k)$ и $S_j(J_1,J_2,...,J_k)$ будут в потоковой зависимости S_i δ S_j, только если существуют значения индексов $(i_1,i_2,...,i_k)$ и $(j_1,j_2,...,j_k)$ такие, что справедливы два условия:

1) $Si(i1,i2,\ldots,ik) \underline{\prec} Sj(j1,j2,\ldots,jk)$;

2) $OUT(S_i(i_1,i_2,\ldots,i_k)) \cap IN(S_j(j_1,j_2,\ldots,j_k)) \neq \varnothing$.

Антизависимость от S_i к S_j (обозначается $S_i \delta^a S_j$) определяется аналогично потоковой, но условие (2) имеет вид $IN(S_i(i_1,i_2,\ldots,i_k)) \cap OUT(S_j(j_1,j_2,\ldots,j_k)) \neq \varnothing$.

Выходная зависимость от S_i к S_j (обозначается $S_i \delta^\circ S_j$) определяется аналогично потоковой, с условием (2): $OUT(S_i(i_1,i_2,...,i_k)) \cap OUT(S_j(j_1,j_2,...,j_k)) \neq \varnothing$.

Граф зависимостей по управлению формируется следующим образом. Каждая вершина, которая может определять последующее исполнение

операторов проверкой логического условия (вершина-распознаватель), имеет не более двух дуг к подчинённым вершинам (т.е. куда передаётся поток управления). Этим дугам сопоставлены атрибуты *T* («истина») и *F* («ложь») – фактически – куда переходить по результатам проверки условия.

Вершина *v* постдоминируется вершиной *w* (*w*≠ *v*) в графе программы (или вершина *w* – обязательный наследник *v*), если каждый путь из *v* в *t* содержит в себе *w* (начальная вершина пути исключается – вершина не постдоминирует сама себя).

Вершина *y зависит по управлению* от *x*, если:

1) существует путь *P* из *x* в *y*, в котором любая вершина *x* (за исключением *x* и *y*) постдоминируется вершиной *y*;
2) вершина *x* не постдоминируется вершиной *y*.

Граф зависимостей по данным и граф зависимостей по управлению вместе составляют граф программных зависимостей (или управляющий граф программы).

Примеры оптимизации

Продвижение констант

Цель оптимизации: Уменьшение избыточных вычислений.

Пути достижения: Продвижение константных значений вниз по коду. Обращение к переменной заменяется её константным значением.

Оригинальный код	После продвижения констант

`int i,n,c;`
`int a[64];`
`n = 64;`
`c = 3;`
`for( i = 0; i < n; i++ )`
`  a[i] = a[i] + c;`
	`int i;`
	`int a[64];`
	`for( i = 0; i < 64; i++ )`
	`  a[i] = a[i] + 3;`

Свертка констант

Цель оптимизации: Уменьшение избыточных вычислений.

Пути достижения: Вычисление константных значений на этапе компиляции.

Оригинальный код	После свертки констант
`int i;` `i = 3*2+1;`	`int i;` `i = 7;`

Распространение копий

Цель оптимизации: Уменьшение числа избыточных переменных содержащих одно и то же значение.

Пути достижения: Использование оригинала вместо копий значения.

Оригинальный код	После распространения копий
`int a[15];`	`int a[15];`

`int t = i * 4;`
`int s = t;`
`int c = 3;`
`cout << a[s];`
`int r;`
`r = t;`
`a[r] = a[r] + c;`
	`int t = i*4;`
	`int c = 3;`
	`cout << a[t];`
	`a[t] = a[t] + c;`

Подстановка операторов

Цель оптимизации: Изменение зависимостей между переменными или улучшение возможностей анализа индексов в циклах.

Пути достижения: Использование переменной заменяется выражением.

Оригинальный код	После подстановки операторов

`int np1 = n + 1;`
`for( i = 0; i < n; i++ )`
`  a[np1] = a[np1] + a[i];`
	`for( i = 0; i < n; i++ )`
	`  a[n+1] = a[n+1] + a[i];`

Прямое преобразование

Цель оптимизации: Снижение стоимости выполнения операций.

Пути достижения: Замена операций эквивалентными с меньшей стоимостью исполнения.

Оригинальный код	После прямого преобразования

`int x;`
`double y;`
`y = y * 2;`
`x = x * 4;`
`x = x / 2;`
	`int x,y;`
	`double y;`
	`y = y + y;`
	`x = x << 2;`
	`x = x >> 1;`

Удаление неиспользуемого кода

Цель оптимизации: Сокращение объема программы за счет удаления неиспользуемого кода.

Пути достижения: Неиспользуемые блоки могут быть обнаружены после использования продвижения констант, анализа неиспользуемых констант.

Оригинальный код	После продвижения констант, свертки константных выражений и удаления неиспользуемого кода

`int x = 0;`
 	
`if( 0 > 1 ) {`
`  function1();`
`}`
 	
`while( x > 0 ) {`
`  function2();`
`}`
 	
`function3();`
	`function3();`

Упрощение булевых выражений в серию переходов

Цель оптимизации: Сокращение вычислений.

Пути достижения: Значение некоторых булевых выражений может быть определено по первому операнду.

Примечание: Подобная интерпретация условного выражения должна быть либо стандартизована в языке (например C/C++), либо выполнятся в случае, когда второе и последующие условия не изменяют переменных и не вызывают функций. В случае если это было предусмотрено языком, то для компилятора подобная интерпретация является обязательной. На примере ниже смысл преобразования показан «логически».

Оригинальный код	После упрощения

`int x,y,c;`
`if( x == 1 && y == 2 )`
`  c = 5;`
	`int x,y,c;`
	`if( x == 1 )`
	`  if( y == 2 )`
	`  c = 5;`

Снижение мощности выражений с индексной переменной

Цель оптимизации: Уменьшение вычисленной сложности некоторых выражений с индексной переменной.

Пути достижения: Замена “сложных” операций “простыми”. Например, замена операций умножения сложением, особенно важно для архитектур, в которых операция умножения выполняется дольше операции сложения.

Исходный текст

```
int n = 64;
int a [64];
void foo (void) {
    for (int i = 0; i < n; i++)
      a [i] = i * 7 ;
}
```

Оригинальный код	Код после снижения мощности
foo:	foo:
pushl %ebp	pushl %ebp
movl %esp, %ebp	movl %esp, %ebp
subl $4, %esp	subl $4, %esp
movl $0, -4(%ebp)	movl $0, -4(%ebp)
.L2:	.L2:
movl -4(%ebp), %eax	movl -4(%ebp), %eax
cmpl n, %eax	cmpl n, %eax
jge .L1	jge .L1
movl -4(%ebp), %eax	movl -4(%ebp), %ecx
movl -4(%ebp), %ecx	movl -4(%ebp), %edx
imull $7, %eax	movl %edx, %eax
movl %eax, a(,%ecx,4)	sall $3, %eax
leal -4(%ebp), %eax	subl %edx, %eax
incl (%eax)	movl %eax, a(,%ecx,4)
jmp .L2	leal -4(%ebp), %eax
.L1:	incl (%eax)
leave	jmp .L2
ret	.L1:

	leave ret

Удаление индексной переменной

Цель оптимизации: Уменьшение числа операций в теле цикла; освобождение регистра, использующегося для хранения индексной переменной.

Пути достижения: Замена условия выхода из цикла

Исходный текст

```
int n = 64;
int a[64];
void foo (void) {
    for (int i = 0; i < n; i++) {
       a[i] = a[i] + 3;
    }
}
```

Оригинальный код	Код после удаления индексной переменной

foo:
pushl %ebp
movl %esp, %ebp
subl $4, %esp
movl $0, -4(%ebp)
.L2:
movl -4(%ebp), %eax
movl $4, %ecx
mull %ecx
movl %eax, %ebx
movl -4(%ebp), %eax
cmpl n, %eax
jge .L1
movl a(%ebx), %eax
addl $3, %eax
movl %eax, a(%ebx)
	foo:
	movl $4, %ecx
	movl n, %eax
	mull %ecx
	leal a, %ecx
	addl %ecx, %eax
	.L2:
	cmpl %eax, %ecx
	jge .L1
	addl $3, (%ecx)
	addl $4, %ecx
	jmp .L2
	.L1:
	leave
	ret

```
        leal -  4(%ebp), %eax
        incl    (%eax)
        jmp     .L2
.L1:
        leave
        ret
```

Раскрутка циклов

Цель оптимизации: Увеличение параллелизма инструкций (увеличение количества инструкций, которые потенциально могут выполняться параллельно), “улучшение” использования регистров и кэша данных.

Пути достижения: Повторение тела цикла несколько раз.

Оригинальный код	После раскрутки цикла

`for (i = 1; i < n-1; i++)`
`  a[i]=(a[i-1]+a[i+1])/2;`
	`for (i = 1; i < n-2; i+=2) {`
	`  a[i]=(a[i-1]+a[i+1])/2;`
	`  a[i+1]=(a[i]+a[i+2])/2;`
	`}`
	`if ((n-2) % 2 == 1)`
	`  a[n-2] =`
	`  (a[n-3]+a[n-1])/2;`

Программная конвейеризация

Цель оптимизации: Увеличение параллелизма инструкций – используется в несуперскалярных процессорах, суперскалярные процессоры самостоятельно конвейеризируют цикл с помощью информации о предсказании переходов.

Пути достижения: Выполнение операций одной итерации цикла разбивается на несколько этапов. Итерации выполняются следующим образом: на i-ой итерации выполняется 1 этап i-ой итерации, 2 этап (i-1)-ой итерации и т.д. Часто это преобразование используется совместно с раскруткой цикла.

Исходный текст

```
int n = 16;
int p [16];
void foo (void) {
    int s = 0;
    for (int i = 0; i < 16; i++)
      s+=abs(((p[i]-p[i-1])+1)<<1);
}
```

Запишем тело этого цикла на псевдоассемблере:

```
1: R1 = Mem (++i)
2: R2 = R0 - R1
3: R3 = ++R2
4: R4 = R3 << 1
5: R5 = abs (R4)
6: R6 = R6 + R5
7: R0 = R1
```

Пусть каждая строка-инструкция выполняется один такт, ограничений на длину команды и количество регистров нет; R0 содержит значение p[0], R6 – значение s.

Ниже приведена таблица инструкций, которые получаются при программной конвейеризации. Исходные инструкции, указанные в одной строке, могут выполняться одновременно. В затемнённых ячейках таблицы указаны номера итераций исходного цикла.

Такт	**Номер инструкции в исходном теле цикла**							**Группа инструкций**
	6	**5**	**4**	**3**	**7**	**2**	**1**	
1							1	Старт-код
2					1	1	2	
3				1	2	2	3	
4			1	2	3	3	4	
5		1	2	3	4	4	5	

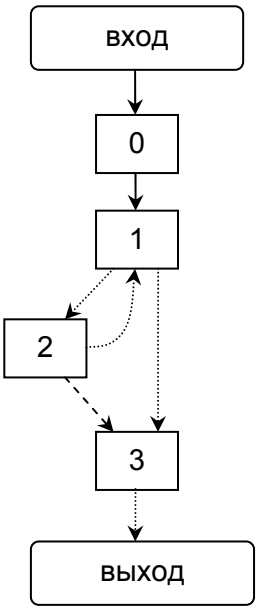
6	1	2	3	4	5	5	6	Итерации цикла (11 итераций)
7	2	3	4	5	6	6	7	
8	3	4	5	6	7	7	8	
9	4	5	6	7	8	8	9	
10	5	6	7	8	9	9	10	
11	6	7	8	9	10	10	11	
12	7	8	9	10	11	11	12	
13	8	9	10	11	12	12	13	
14	9	10	11	12	13	13	14	
15	10	11	12	13	14	14	15	
16	11	12	13	14	15	15	16	
17	12	13	14	15	16	16		Код зачистки
18	13	14	15	16				
19	14	15	16					
20	15	16						
21	16							

Вынесение условных выражений за пределы цикла

Цель оптимизации: Уменьшение накладных расходов на проверку условия с инвариантной относительно цикла переменной.

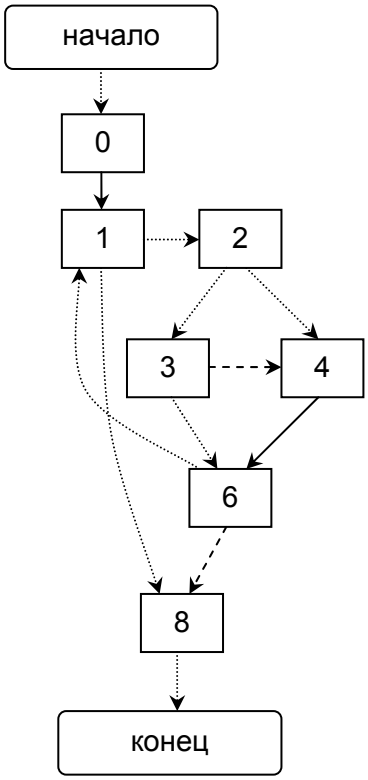
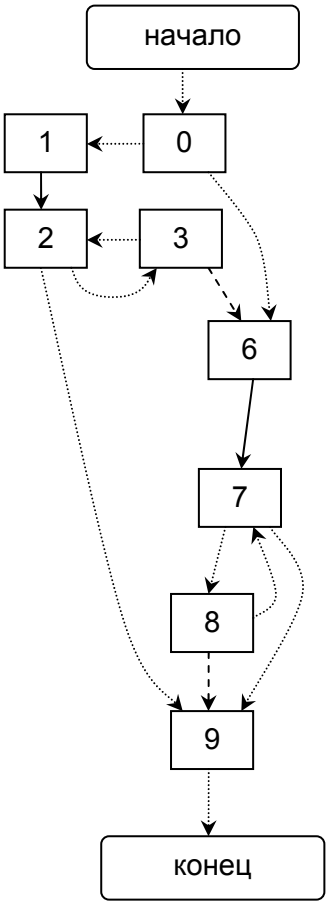
Пути достижения: Перенос цикла в ветви условного оператора.

	Граф потока управления для цикла

`void foo (void) {`
`0:  int i;`
`0:  i = 0;`
`1:  while (i < n) {`
`2:  /*тело цикла*/`
`3:  }`
`}`
	

Исходный код	Исходный код после вынесения условного выражения

`int n = 64;`
`int a [64];`
`int b [64];`
`void foo (void) {`
`0:  int i = 0;`
`0:  int c;`
`1:  while (i < n){`
`2:  a[i]=a[i]+4;`
`2:  if (c<10)`
`3:  b[i]=a[i]*b[i-1];`
`2:  else`
`4:  b[i]=a[i]+6;`
`6:  i++;`
`8:  }`
`}`
	`int n = 64;`
	`int a [64];`
	`int b [64];`
	`void foo (void) {`
	`0:  int i = 0;`
	`0:  int c;`
	`0:  if (c < 10)`
	`2:  while (i < n) {`
	`3:  a [i] = a[i]+4;`
	`3:  b[i]=a[i]*b[i-1];`
	`3:  i++;`
	`}`
	`0:  else`
	`7:  while (i < n) {`
	`8:  a [i] = a[i]+4;`
	`8:  b[i]=a[i]+6;`
	`8:  i++;`
	`9:  }`
	`}`

Исходный граф потока управления	Граф потока управления после вынесения условного выражения
	

Вынесение первых и последних итераций

Цель оптимизации: Удаление зависимостей, вызванных несколькими первыми или последними итерациями.

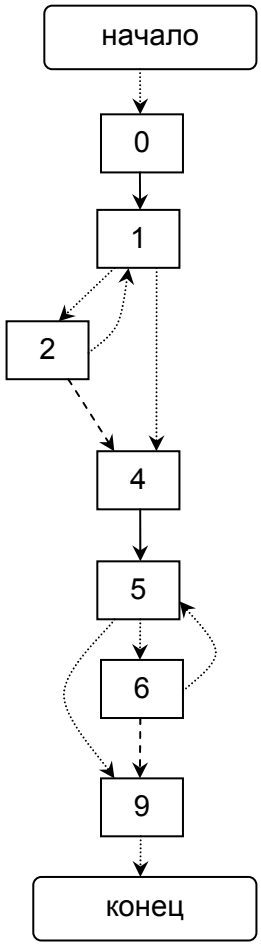
Пути достижения: Вынесение нескольких первых или последних итераций за пределы цикла.

Исходный текст	Код после вынесения первой итерации второго цикла

`int n = 64;`
`int a [64];`
`int b [64];`
`void foo (void) {`
`0: int i = 3;`
`1: while (i < n) {`
	`int n = 64;`
	`int a [64];`
	`int b [64];`
	`void foo (void) {`
	`0: int i = 2;`
	`0: if (i <= 2)`

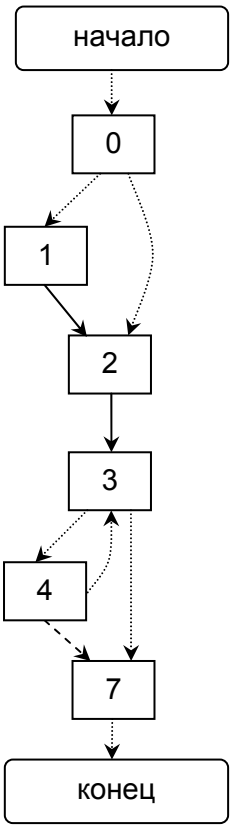
```
2:            b [i] = b [i] + b [2];
2:            i++;
4:         }
           i = 2;
5:         while (i < n) {
6:            a [i] = a [i] + 3;
6:            i++;
9:         }
       }
```

Исходный граф потока данных




```
1:            a [i] = a [i] + 3;
2:         i = 3;
3:         while (i < n) {
4:            a [i] = a [i] + 3;
4:            b [i] = b [i] + b[2]
4:            i++;
7:         }
       }
```

Граф потока данных после вынесения первой итерации первого цикла



Оптимизация хвостовых вызовов

Цель оптимизации: Уменьшить накладные расходы при вызове функций.

Пути достижения: Замена вызова (call) на безусловный переход (jmp).

Исходная программа

```
int bar (int a) {
     printf ("bar!");
     if (a == 0)
          return -1;
     return a;
}
```

```
int foo (int a) {
printf ("foo!");
if (a == 0)
return 0;
return bar (a);
}
```

Оптимизация вызовов отключена

```
bar:
     pushl    %ebp
     movl     %esp, %ebp
     …
     leave
     ret
foo:
     pushl    %ebp
     movl     %esp, %ebp
     …
     subl     $12, %esp
     pushl    8(%ebp)
     call     bar
     addl     $16, %esp
     …
     leave
     ret
```

Оптимизация вызовов включена

```
bar:
     pushl    %ebp
     movl     %esp, %ebp
     …
     leave
     ret
foo:
     pushl    %ebp
     movl     %esp, %ebp
     …
     movl     8(%ebp), %eax
     movl     %eax, 8(%ebp)
     leave
     jmp      bar
     …
     leave
     ret
```

Встраивание функций (Inline)

Цель оптимизации: Снижение накладных расходов на вызовы функций (за счет увеличения размера).

Пути достижения: Копирование тела встраиваемой функции в вызывающую функцию. Конкретная реализация определяется компилятором. Обычно есть ряд ограничений на разрастание кода и на присутствие в коде различных управляющих структур, например вызова других функций.

Оригинальный код	После встраивания

`void bar (int* a, int i) {`
`  a[i]=a[i]+3;`
`}`
 	
`void foo (void) {`
`  int i;`
`  for (i = 3; i < n; i++)`
`  bar (a, i);`
`}`
	`void foo (void) {`
	`  int i;`
	`  for (i = 3; i < n; i++)`
	`  a[i]=a[i]+3;`
	`}`

Приложение А. Установка GCC

Кратко опишем процесс установки GCC с использованием исходных кодов. Более подробную информацию об установке можно получить из документа “GCC Installation Guide” (http://gcc.gnu.org/install/).

Получение дистрибутива.

Как получить последнюю версию дистрибутива компилятора можно узнать по адресу http://gcc.gnu.org/releases.html. На момент написания данного пособия последняя версия компилятора была 3.3.3. Существует два основных варианта дистрибуции: компилятор в полном объёме (со всеми языками, которые входят в стандартную поставку), или отдельно ядро компилятора и ряд пакетов поддержки входных языков. Для определённости, далее предполагается использование первого варианта, в этом случае файл с исходными кодами компилятора имеет имя gcc-3.3.3.tar.bz2. Для начала работ по установке нужно извлечь содержимое архива:

```
# tar -jxf gcc-3.3.3.tar.bz2
```

В результате выполнения этой команды в текущем каталоге появится каталог `gcc-3.3.3` с исходными кодами GCC.

Конфигурирование GCC

Перед началом установки необходимо создать каталог для хранения объектных модулей и исполняемых файлов. Рекомендуется не использовать дерево каталогов gcc, которое получается в результате распаковки архива (т.е. в нашем случае, `gcc-3.3.3`). Создать каталог можно следующим образом:

```
# mkdir objdir
# cd objdir
```

Далее необходимо запустить сценарий конфигурации:

```
# ../gcc-3.3.3/configure --prefix=somedir
        --enable-languages=somelang
```

Опция `--prefix` указывает каталог, куда будет установлен GCC (по умолчанию это `/usr/local`). Опция `--enable-languages` содержит список необходимых языков, разделенных запятой. Стандартная поставка поддерживает следующие языки: `ada`, `c`, `c++`, `f77`, `java`, `objc`.

Компиляция

Для запуска компиляции необходимо выполнить следующую команду:

```
# make bootstrap
```

Тестирование

Для проверки работоспособности и корректности компилятора можно выполнить тестирование, но этот шаг необязателен. Описание тестирования можно найти в документе “GCC Testing” (http://gcc.gnu.org/install/test.html). Выполнение тестирования:

```
# make -k check
```

Установка

```
# make install
```

Происходит копирование файлов в указанную директорию *somedir* и создание справочной документации.

Приложение Б. Использование GCC

Во время работы GCC обычно выполняет препроцессирование, компиляцию, ассемблирование и линковку. «Общие опции» позволяют остановить этот процесс на промежуточной стадии. Например, при указании опция `-c` не запускается линкер. Тогда вывод состоит их объектных файлов, порожденных ассемблером.

Другие опции передаются на одну из стадий обработки. Одни опции управляют препроцессором, другие самим компилятором; имеются опции, управляющие ассемблером и линкером; большинство из них не описано здесь, поскольку редко требуется использовать какую-нибудь из них.

Можно указывать опции и другие аргументы в любом порядке. По большей части, используемый порядок не имеет значения. Но, порядок важен, когда используется несколько опций одного вида. Например, если указана опция `-L` (см. ниже) больше чем один раз, директории просматриваются в порядке указания.

Многие опции имеют длинные имена, начинающиеся с `-f` или с `-W`. Большинство из них имеет положительную и отрицательную формы; отрицательной формой `-ffoo` будет `-fno-foo`.

Общие опции

`-x` *`язык`*

Указывает, какой язык использовать для заданного входного файла. Возможные значения опции: `c`, `c-header`, `cpp-output`, `c++`, `c++-header`, `c++-cpp-output`, `objective-c`, `objective-c-header`, `objc-cpp-output`, `assembler`, `assembler-with-cpp`, `ada`, `f77`, `f77-cpp-input`, `ratfor`, `java`, `treelang`.

```
$ gcc -x java test.java
```

`-c`

Отключает линковщик. Исходные файлы компилируются или ассемблируются, но не линкуются. Конечный вывод происходит в форме объектного файла для каждого исходного файла. По умолчанию, имя объектного файла формируется из имени исходного файла заменой суффикса («.c», «.cpp» и т. д.) на «.o».

```
$ gcc -c somelib.c
```

`-S`

Остановиться после компиляции, не ассемблировать. Вывод производится в форме файла с ассемблерным кодом для каждого не ассемблерного входного файла. По умолчанию, имя файла с ассемблерным кодом делается из имени исходного файла заменой суффикса («.c», «.cpp» и.т. д.) на «.s».

```
$ gcc -S sibcall.c
```

`-E`

Остановиться после стадии препроцессирования, не запускать собственно компилятор. Вывод делается в форме препроцессированного исходного кода, который посылается на стандартный вывод.

```
$ gcc -E test.cpp
```

`-o` *`имя_файла`*

Поместить вывод в файл *`имя_файла`*. Эта опция применяется вне зависимости от вида порождаемого файла, есть ли это выполнимый файл, объектный файл, ассемблерный файл или препроцессированный C код. Поскольку указывается только один выходной файл, нет смысла использовать `-o` при компиляции более чем одного входного файла, если не порождается выполнимый файл.

Если `-o` не указано, по умолчанию выполнимый файл помещается в a.out, объектный файл для `source.suffix` – в `source.o`, его ассемблерный код в `source.s` и все препроцессированные C файлы – в стандартный вывод.

```
$ gcc -o program source1.c source2.c object1.o
object2.o
```

`-pipe`

Использовать неименованные программные каналы вместо временных файлов для коммуникации между различными стадиями компиляции.

Опции оптимизации и генерации отладочной информации

`-g`

Порождает отладочную информацию в родном формате операционной системы (stabs, COFF, XCOFF или DWARF). GDB (отладчик) может работать с этой отладочной информацией.

`-O1, -O2, -O3, -Os`

Оптимизировать. Этими флагами устанавливается то, какие проходы оптимизации будут выполнены.

Флаг `-Os` включает оптимизацию размера.

`-O0`

Не оптимизировать.

Опции поиска каталогов и подключения библиотек

`-l`*`library`*

Ищет при линковке библиотеку с именем *`library`*. Есть различие в том, где в командной строке указана эта опция: линкер ищет обрабатываемые библиотеки и объектные файлы в порядке, в котором они указаны. Таким

образом, `foo.o -lz bar.o` ищет библиотеку `z` после файла `foo.o`, но перед `bar.o`. Если `foo.o` ссылается на функции в `z`, эти функции не могут быть загружены.

Линкер просматривает стандартный список каталогов в поисках библиотеки, которая, фактически, является файлом с именем `lib`*`library`*`.a`. Затем линкер использует этот файл так, как будто бы он был точно специфицирован по имени.

Директории, в которых ищет линкер, включают несколько стандартных системных каталогов, плюс любые каталоги, которые определены с помощью опции `-L` (см. ниже).

Обычно файлы, обнаруженные этим способом являются архивными файлами, чьи элементы – объектные файлы. Линкер обрабатывает архивный файл, просматривая его в поиске элементов, определяющих символы, на которые были ссылки, но которые до сих пор не определялись. Но, если оказывается, что обнаруженный файл – обычный объектный файл, то он линкуется в обычном порядке. Единственное различие между использованием опции -l и указанием имени файла в том, что `-l` добавляет `lib` и `.a` к *`library`* и ищет в нескольких каталогах.

`-I`*`directory`*

Добавляет каталог *`directory`* в начало списка каталогов, используемых для поиска заголовочных файлов. Опцию можно использовать для подмены системных заголовочных файлов, подставляя собственные версии, поскольку эти каталоги просматриваются до каталогов системных заголовочных файлов. Если использована более чем одна опция `-I`, каталоги просматриваются в порядке слева направо; стандартные системные каталоги просматриваются в последнюю очередь.

`-L`*`directory`*

Добавляет каталог *`directory`* в начало списка каталогов, используемых для поиска библиотек. Работа опции аналогична `-I`.

Пример компиляции

Пусть необходимо скомпилировать программу `test`. Уже имеются объектные модули `foo.o` и `bar.o`, причем модуль `bar.o` использует библиотеку `z`. Библиотека `z`, расположена в каталоге `/usr/home/project/lib`. Имеется так же файл `test.c`, использующий заголовочные файлы из `/usr/home/project/include`. Командная строка для запуска GCC для описанного случая будет выглядеть следующим образом:

```
$ gcc -o test -I/usr/home/project/include
        -L/usr/home/project/lib test.c foo.o -lz bar.o
```

В результате выполнения будет создан исполняемый файл с именем `test`

Приложение B. Каталоги GCC

Корневой каталог

`INSTALL`

Документация по конфигурированию и установке GCC.

`boehm-gc`

Boehm (имя собственное) сборщик мусора – часть Java Runtime Library.

`config`

Дополнительные файлы для конфигурирования (добавляют некоторые флаги в Makefile'ы в зависимости от архитектуры процессора и операционной системы); обычно нужны для запуска autoconf.

`contrib`

Скрипты, используемые разработчиками GCC. См. contrib.

`fastjar`

Реализация команды jar; используется с Java front end.

`gcc`

Основной каталог gcc. См. Подкаталог gcc.

`include`

Заголовочные файлы для libiberty, библиотеки поддержки, используемой компилятором.

`libf2c`

The Fortran runtime library.

`libffi`

Библиотека libffi – часть Java runtime library.

`libiberty`

Используется для переносимости, а так же содержит некоторые основные структуры данных и алгоритмы; фактически, поддержка необходимых операций.

`libjava`

The Java runtime library.

`libobjc`

The Objective-C runtime library.

`libstdc++-v3`

The C++ runtime library.

`maintainer-scripts`

Скрипты разработчиков. См. maintainer-scripts.

`zlib`

Библиотека сжатия, используется Java front end'ом и в Java runtime library.

contrib

`analyze_brprob`

Этот скрипт используется для предсказания ветвлений. Применяются эвристический и hitrate подходы.

`compare_tests`

Скрипт предназначен для автоматического тестирования некоторой утилиты и создания отчетов об этом тестировании.

`convert_to_f2c`

Переименовывает некоторые файлы из libg2c.

`convert_to_g2c`

Переименовывает некоторые файлы из libg2c.

`download_f2c`

Загружает tarball netlib.

`gcc_build`

Автоматическая загрузка и сборка GCC.

`gcc_update`

Обновляет локальное дерево CVS с репозитория GCC.

`gccbug.el`

Пересылка отчета об ошибке gnats.

`gennews`

Автоматическое создание NEWS-файлов из он-лайн Release Notes.

`newcvsroot`

Заменяет все файлы в CVS/Root и CVS/Repository.

`test_installed`

Запуск тестов.

`test_summary`

Формирует отчеты о тестировании и пересылает их.

`texi2pod.pl`

Преобразует из texi в pod.

`warn_summary`

Сбор статистики при сборке GCC.

Подкаталог gcc

`'language'`

Подкаталоги с описанием языков. Подробнее см. 'language' (language заменяется на имя языка: cp, java, ada, f, objc и т.д.).

`config`

Файлы конфигурации для поддерживаемых архитектур и операционных систем. Подробнее см. config.

`doc`

Документация в формате Texinfo.

`fixinc`

Поддержка корректировки системных заголовочных файлов; для обеспечения работы с GCC.

`ginclude`

Системные заголовочные файлы устанавливаемые GCC.

`intl`

libintl.

`po`

Поддержка различных языков.

`testsuite`

Тесты GCC. Тестовая подсистема, поддерживаемая с помощью комплекса dejagnu.

'language'

`config-lang.in`

Описание языка. Подробнее: стр. 30 GCC Internals.

`Make-lang.in`

`lang-options.h`

Опции коммандной строки для fron end'a.

`lang-specs.h`

config

`'machine'`

Каталог с описанием архитектуры. Содержит machine.md, заголовки (h-файлы) и файлы с исходным текстом. В данных файлах с помощью более-менее стройного метода описания gcc описываются разнообразные процессоры. Естественно, самые «продвинутые» оптимизации описываются достаточно сложно и не в полной мере.

Приложение Г. LEX (FLEX)

Лексический анализатор для GCC Front end генерируется при помощи Lex (FLEX). Lex предназначен для создания лексических анализаторов, которые могут использоваться вYacc.

Входной язык содержит описания лексем в терминах регулярных выражений. Результатом работы LEX'a является программа на языке Си, которая читает файл yyin (обычно это стандартный ввод) и выделяет из него последовательности символов (лексемы), соответствующие регулярным выражениям.

Рассмотрим способы записи регулярных выражений во входном языке Lex'a. Алфавит Lex'a совпадает с алфавитом используемой ЭВМ. Символ из алфавита, естественно, представляет регулярное выражение из одного символа. Специальные символы (в том числе + - * ? ( ) [ ] { } | / \ ^ $. < >) записываются после специального префикса \. Кроме того, применимы все традиционные способы кодирования символа в языке С. Символы и цепочки можно брать в кавычки:

Примеры:

- a "a" \a – три способа кодирования символа a
- \n \t \007 "continue"
- Имеется возможность задания класса символов:
    - [0-9] или [0123456789] – любая цифра
    - [A-Za-z] – любая буква
    - [^0-7] – любой символ, кроме восьмеричных цифр
    - . – любой символ, кроме \n

Грамматика для записи регулярных выражений (в порядке убывания приоритета):

<p> : <p>* – повторение 0 или более раз

<p> : <p>+ – повторение 1 или более раз

<p> : <p>? – необязательный фрагмент

<p> : <p><p> – конкатенация

<p> : <p>{m,n} – повторение от m до n раз

<p> : <p>{m} – повторение m раз

<p> : <p>{m,} – повторение m или более раз

<p> : ^<p> – фрагмент в начале строки

<p> : <p>$ – фрагмент в конце строки

<p> : <p>|<p> – любое из выражений

<p> : <p>/<p> – первое выражение, если за ним следует второе

<p> : (p) – скобки, используются для группировки

Примеры:

[A-Za-z]([A-Za-z0-9]{0,7}) – идентификатор (имя) в языке C

^#" "*define – начало #define в языке C

Программа на входном языке Lex состоит из трех частей, разделенных символами %%:

```
Описания
%%
Правила
%%
Программы
```

Раздел описаний содержит определения макросимволов (метасимволов) в виде:

```
ИМЯ ВЫРАЖЕНИЕ
```

Если в последующем тексте в регулярном выражении встречается {ИМЯ}, то оно заменяется ВЫРАЖЕНИЕМ. Если строка описаний начинается с пробелов или заключена в скобки %{ ... }%, то она просто копируется в выходной файл.

Раздел правил содержит строки вида

```
ВЫРАЖЕНИЕ {ДЕЙСТВИЕ}
```

ДЕЙСТВИЕ – это фрагмент программы на С, который выполняется тогда, когда обнаружена цепочка, соответствующая ВЫРАЖЕНИЮ. Действие, указанное в начале раздела без выражения, выполняется до начала разбора. Lex делает попытку выделить наиболее длинную цепочку из входного потока. Если несколько правил дают цепочку одинаковой длины, применяется первое правило. Так, при разборе по следующим правилам для цепочки "123" будет применено первое правило, а для цепочки "123." – третье:

```
[0-9]+
(\+|\-)?[0-9]+
(\+|\-)?[0-9]+"."[0-9]+
```

Если ни одно из правил не удастся применить, входной символ будет скопирован в yyout. Если это нежелательно, в конец правил можно добавить, например, строки:

```
. {/* Ничего не делать */}
\n { }
```

Раздел программ может содержать произвольные тексты на С и целиком копируется в выходной файл. Обычно здесь записывается функция yywrap(), которая определяет, что делать при достижении автоматом конца

входного файла. Ненулевое возвращаемое значение приводит к завершению разбора, нулевое – к продолжению (перед продолжением, естественно, надо открыть какой-нибудь файл как yyin).

Интерпретатор таблиц конечного автомата имеет имя yylex(). Автомат прекращает работу (происходит возврат из функции yylex()), если в одном из действий выполнен оператор return (результат yylex() будет совпадать с указанным в операторе) или достигнут конец файла и значение yywrap() отлично от 0 (результат yylex() будет равен 0).

Традиционный пример входной программы для Lex'a – подсчет числа слов и строк в файле:

```
/***************** Раздел определений ***************/
/*   NODELIM означает любой символ, кроме
     разделителей слов  */
NODELIM    [^" "\t\n]
           int l,  /* Число строк */
               w,  /* Число слов */
               c;  /* Число символов */
%% /***************** Раздел правил *****************/
            { l=w=c=0;         /* Инициализация */     }
{NODELIM}+  { w++; c+=yyleng; /* Слово */              }
\n          { l++;            /* Перевод строки */    }
.           { c++;            /* Остальные символы */ }
%% /*************** Раздел программ *****************/
main() { yylex(); }
yywrap() {
  printf(" Lines - %d  Words - %d Chars - %d\n",
          l, w, c );
  return( 1 );
}
```

Внутри действий в правилах можно использовать некоторые специальные конструкции и функции Lex'a:

`yytext`

– указатель на отождествленную цепочку символов, терминированную нулем;

`yyleng`

– длина этой цепочки

`yyless(n)`

– вернуть последние n символов цепочки обратно во входной поток;

`yymore()`

– считать следующие символы в буфер yytext после текущей цепочки

`yyunput(c)`

– поместить байт во входной поток

`ECHO`

– копировать текущую цепочку в yyout

Приложение Д. YACC (BISON)

Программа *YACC (Yet Another Compiler Compiler)* предназначена для построения синтаксического анализатора контекстно-свободного языка. Анализируемый язык описывается с помощью грамматики в виде, близком форме Бэкуса-Наура (БНФ). Результатом работы YACC'а является программа на Си, реализующая восходящий LALR(1) распознаватель.

Структура YACC-программы

YACC-программа состоит из трех секций, разделенных символом %% – секции описаний, секции правил, в которой описывается грамматика, и секции программ, содержимое которой просто копируется в выходной файл. Пример простейшей программы на YACC'е:

```
%token name
%start e
%%
e : e '+' m | e '-' m | m ;
m : m '*' t | m '/' t | t ;
t : name | '(' e ')' ;
%%
```

Секция правил содержит информацию о том, что символ name является лексемой (терминалом) грамматики, а символ e – ее начальным нетерминалом.

Грамматика записана обычным образом – идентификаторы обозначают терминалы и нетерминалы, символьные константы типа '+' '-' – терминалы. Символы : | ; принадлежат метаязыку и читаются «есть по определению», «или» и «конец правила» соответственно.

Разрешение конфликтов

Написанная грамматика (она обладает свойством LALR(1)) задает язык арифметических формул, в котором приоритет '*' и '/' выше приоритета '+' и '-', а все операции левоассоциативны. Для указания этих свойств языка в грамматику введены дополнительные нетерминалы m, и t. Другая грамматика этого языка:

```
e : e '+' e | e '-' e | e '*' e |
    e '/' e | '(' e ')' | name ;
```

не однозначна (и, следовательно, не LALR(1)). Попытка применить YACC для анализа данной грамматики приведет к многочисленным неразрешенным конфликтам типа сдвиг/свертка (shift/reduce) в построенном автомате. Если рассмотреть конфликты более подробно, выясняется, что в каждом случае можно однозначно выбрать между сдвигом или сверткой, основываясь на приоритетах операций и порядке выполнения равноприоритетных операций слева направо. (Аналогично простому и операторному предшествованию).

YACC позволяет дополнить грамматику информацией такого рода и получить бесконфликтный распознаватель:

```
%token name
%left '+' '-'
%left '*' '/'
%%
e : e '+' e | e '-' e | e '*' e | e '/' e
  | '(' e ')' | name ;
%%
```

Предложения `%left`, `%right` и `%nonassoc` в секции описаний приписывают всем перечисленным за ними символам одинаковый приоритет и соответствующее значение ассоциативности. (Отсутствие ассоциативности означает недопустимость выражений вида a @ b @ c) Приоритет увеличивается сверху вниз для каждого нового предложения.

LALR(1)-конфликты сдвиг/свертка или свертка/свертка разрешаются выбором более приоритетного действия. Приоритет сдвига равен приоритету считываемой лексемы. Приоритет свертки равен приоритету самой правой лексемы в правиле, по которому производится свертка. Можно также явно указать приоритет свертки, написав `%prec <лексема>` справа от правила.

Добавить в язык формул операцию унарного минуса, более приоритетную, чем бинарные операции, можно следующим образом:

```
%token name
%left '+' '-'
%left '*' '/'
%left UMIN
%%
e : e '+' e | e '-' e | e '*' e | e '/' e
  | '(' e ')' | name ;
e : '-' e %prec UMIN ;
%%
```

Фиктивная лексема UMIN используется только для задания приоритета свертки по правилу `e : '-' e` ;

Итак, YACC разрешает конфликты (если они возникнут) по следующим правилам:

- если приоритеты альтернативных действий определены и различны, то выполняется действие с б*о*льшим приоритетом;
- если приоритеты действий определены и одинаковы, то в случае левой ассоциативности выбирается свертка, в случае правой ассоциативности – сдвиг, если они неассоциативны – создается ошибочная ситуация;

- иначе (приоритет хотя бы одной из альтернатив не специфицирован) в случае конфликта сдвиг/свертка выполняется сдвиг, а в случае конфликта свертка/свертка – свертка по правилу, определенному выше по тексту в описании грамматики, в обоих случаях YACC сообщает о неразрешенном конфликте в этом состоянии.

Отметим, что для конфликтной грамматики с правилами

```
s : if '(' e ')' s
| if '(' e ')' s else s
;
```

предпочтение сдвига «правильно» разрешает конфликт при разборе выражения

```
if( e ) if( e ) s else s
```

else будет отнесено к ближайшему if'у, как и принято в алголоподобных языках.

Для конфликтной грамматики арифметических формул, эти правила приводят к вычислению выражения справа налево без учета приоритетов операций.

Семантические действия

С каждым правилом грамматики может быть связано действие, которое будет выполнено при свертке по данному правилу. Оно записывается в виде заключенной в фигурные скобки последовательности операторов языка Си, расположенной после правой части соответствующего правила.

```
stnt : IF '(' expr ')' stnt          { if_ctr++; }
          | WHILE '(' expr ')' stnt  { while_ctr++; }
          | assign_st                { ass_ctr++; };
```

В этом примере действие if_ctr++ будет выполнено после разбора всего оператора if. При необходимости выполнить семантические действия,

например, сразу после вычисления выражения expr, можно поместить их между символами правой части правила.

```
statement: IF '(' expr { action_1 } ')'
                statement { action_2 } ;
```

В этих случаях YACC автоматически преобразует грамматику, вводя дополнительные нетерминалы и соответствующие им правила с пустой правой частью. При их свертке и будут выполнены действия, расположенные между символами исходной грамматики.

```
stnt: IF '(' expr void_1 ')' stnt { action_2 } ;
void_1:    { action_2 } ;
```

Семантический стек

Для естественного обмена данными между действиями, каждый терминал или нетерминал может иметь значение. Для доступа к нему в действиях используются псевдопеременные $$ – значение левого символа правила, $<i> – значение i-ого символа правой части правила (символы нумеруются слева направо, начиная с 1). Другими словами, кроме обычного стека состояний построенный YACC'ом анализатор содержит «семантический» стек, содержащий значения символов. Значения имеют тип YYSTYPE, который по умолчанию определяется как int. Действие

```
expr : expr '+' expr { $$ = $1 + $3; } ;
```

может быть использовано в интерпретаторе формул, в котором значение нетерминала «выражение» есть его вычисленное значение.

Если для правила не указано никакого действия, или действие не содержит упоминания псевдопеременной $$, то значение левой части правила становится равным значению первого символа правой части, т. е. неявно выполняется действие { $$ = $1; }. Значение очередной лексемы копируется из переменной int yylval, в которую его обычно заносит сканер.

Различные символы грамматики могут иметь значения разных типов.

Для этого следует определить тип YYSTYPE как union и специфицировать тип терминалов и нетерминалов в разделе описаний. При этом будет осуществляться контроль типов при использовании псевдопеременных, а обращение к ним будет транслироваться в обращение к соответствующему полю union.

```
%{
#define YYSTYPE yys
typedef union {
    int      intval;
    long     longval;
    nodeptr *ptrval;
} yys;
%{
%token <intval> ICONST
%token <intval> LCONST
%type  <ptrval> expr
```

Если в качестве внутреннего представления программы используется дерево, удобно иметь в качестве значения нетерминала указатель на соответствующий ему узел дерева.

Кодировка лексем и интерфейс

Файл, порождаемый YACC'ом в процессе работы, содержит таблицы LALR(1)-анализатора и Си-текст функции int yyparse(void), реализующей интерпретатор таблиц и семантические действия. Для запуска парсера достаточно вызвать эту функцию. В случае успешного разбора она возвращает 0, в случае ошибки – 1.

Для получения очередной лексемы парсер вызывает функцию int yylex(void). Она должна возвратить код лексемы и поместить ее значение в переменную YYSTYPE yylval.

Код лексемы – положительное целое число. Лексемам, заданным в виде символьных констант, соответствует их код в наборе символов ЭВМ (обычно ASCII), лежащий в диапазоне 0..255. Лексемам, имеющим символические имена, присваиваются коды начиная с 257.

Выходной файл содержит операторы #define, определяющие имена лексем как их коды. Если имена лексем требуются в других файлах, следует указать ключ -d при запуске YACC'a, и он продублирует эти определения в файле y.tab.h. Этот файл следует включить в другие файлы программы (например, сканер), использующие коды лексем.

Обработка ошибок

Если анализируемое предложение не соответствует языку, то в некоторый момент возникнет ошибочная ситуация, т. е. парсер окажется в состоянии, в котором не предусмотрено ни сдвига, ни свертки для полученной ошибочной лексемы. Обычная реакция парсера – вызов функции void yyerror(const char *) с аргументом "Syntax error" и завершение работы – возврат из функции yyparse с значением 1. Реализация функции yyerror возлагается на пользователя, и он может попытаться организовать в ней выдачу более разумной диагностики (при использовании YACC-парсера это не является тривиальной задачей).

Во многих случаях желательно как-нибудь продолжить разбор. Для восстановления после ошибки YACC содержит следующие средства. Имеется специальная лексема с именем error, которая может употребляться в грамматике. При возникновении ошибки устанавливается флаг ошибки, вызывается функция yyerror, а затем из стека состояний удаляются элементы, пока не встретится состояние, допускающее лексему error. При обнаружении такого состояния выполняется сдвиг, соответствующий лексеме error в этом состоянии и разбор продолжается. Если при установленном флаге ошибки снова возникает ошибочная ситуация, то для

избегания многократных сообщений yyerror не вызывается, а ошибочная лексема игнорируется. Флаг ошибки сбрасывается только после трех успешно считанных лексем.

Специальными действиями в правилах, обрабатывающих ошибочные ситуации можно более активно вмешиваться в этот процесс.

yyerrok() – сбрасывает флаг ошибки

yyclearin() – удаляет просмотренную вперед ошибочную лексему

Макро YYERROR явным образом возбуждает ошибочную ситуацию.

Пример:

```
statement :  ....
          | error ';'
```

при возникновении ошибки внутри statement продолжение разбора возможно только начиная с ';' – в результате будут пропущены все лексемы до точки с запятой, которая затем будет свернута в нетерминал statement.

Приложение E. Lex.l

```
%{ /* -*- c -*- = mode for emacs editor
/*
   VMK lexical analysis
*/

#include <stdio.h>
#include <memory.h>
#include "ansidecl.h"
#include "config.h"
#include "system.h"

/* Avoid poisoned malloc problem.  */
#undef IN_GCC

#include "config.h"
#include "diagnostic.h"
#include "tree.h"

/* Token defs.  */
#include "vmk.h"
#include "parse.h"

%}

%%
constant{ yylval.ival = 0; return ARG_NUMBER; }
unary      { yylval.ival = 1; return ARG_NUMBER; }
binary   { yylval.ival = 2; return ARG_NUMBER; }
ternary { yylval.ival = 3; return ARG_NUMBER; }
function{ return FUNCTION; }
is       { return IS; }

firstly { return FIRST; }
first      { yylval.ival = 0; return PARAM; }
second   { yylval.ival = 1; return PARAM; }
third      { yylval.ival = 2; return PARAM; }

zero       { yylval.ival = 0; return ZERO_NUMBER; }

one      { yylval.ival = 1; return SIMPLE_NUMBER; }
two      { yylval.ival = 2; return SIMPLE_NUMBER; }
three      { yylval.ival = 3; return SIMPLE_NUMBER; }
four       { yylval.ival = 4; return SIMPLE_NUMBER; }
five       { yylval.ival = 5; return SIMPLE_NUMBER; }
six      { yylval.ival = 6; return SIMPLE_NUMBER; }
seven      { yylval.ival = 7; return SIMPLE_NUMBER; }
```

```
eight       { yylval.ival = 8; return SIMPLE_NUMBER; }
nine        { yylval.ival = 9; return SIMPLE_NUMBER; }

ten      { yylval.ival = 10; return TENS_NUMBER; }
eleven   { yylval.ival = 11; return TENS_NUMBER; }
twelve   { yylval.ival = 12; return TENS_NUMBER; }
thirteen{ yylval.ival = 13; return TENS_NUMBER; }
fourteen{ yylval.ival = 14; return TENS_NUMBER; }
fifteen  { yylval.ival = 15; return TENS_NUMBER; }
sixteen  { yylval.ival = 16; return TENS_NUMBER; }
seventeen  { yylval.ival = 17; return TENS_NUMBER; }
eighteen{ yylval.ival = 18; return TENS_NUMBER; }
nineteen{ yylval.ival = 19; return TENS_NUMBER; }

twenty   { yylval.ival = 20; return COMPOSITE_NUMBER; }
thirty   { yylval.ival = 30; return COMPOSITE_NUMBER; }
fourty   { yylval.ival = 40; return COMPOSITE_NUMBER; }
fifty       { yylval.ival = 50; return COMPOSITE_NUMBER; }
sixty       { yylval.ival = 60; return COMPOSITE_NUMBER; }
seventy  { yylval.ival = 70; return COMPOSITE_NUMBER; }
eighty   { yylval.ival = 80; return COMPOSITE_NUMBER; }
ninety   { yylval.ival = 90; return COMPOSITE_NUMBER; }

hundred  { return HUNDRED; }
and         { return AND; }
then        { return THEN; }
argument{ return ARGUMENT; }
arg      { return ARGUMENT; }


plus        { return PLUS; }
minus       { return MINUS; }
mul      { return MUL; }
div      { return DIV; }

[0-9]+   { yylval.ival = atoi( yytext ); return NUM; }
[a-zA-Z]+  { yylval.name = yytext; return NAME; }
[ \n]       ;
.   { return yytext[0]; }
%%

int yywrap()
{
  return 1;
}
```

Приложение Ж. parce.y

```
/* Grammer file for compiler for toy language.  */

%{
#include "stdio.h"
#include "config.h"
#include "system.h"
#include "tree.h"
#include "vmk.h"

void yyerror (const char *error_message ATTRIBUTE_UNUSED);
int yylex(void);

%}

%union {
tree exp;          /* Tree node representing value. */
int ival;          /* Integer value for constant or arg */
char *name;             /* Name of function */
}

%token_table
%token <ival> NUM       /* Decimal constant. */
%token <name> NAME      /* Function name.  */
%token PLUS             /* Plus operator */
%token MINUS            /* Minus operator */
%token MUL         /* Multiple operator */
%token DIV         /* Divide operator */
%token FIRST
%token THEN

/* Digits */
%token <ival> ZERO_NUMBER
%token <ival> SIMPLE_NUMBER
%token <ival> TENS_NUMBER
%token <ival> COMPOSITE_NUMBER
%token HUNDRED
%token AND

%token <ival> ARG_NUMBER
%token <ival> PARAM
%token FUNCTION
%token IS
%token ARGUMENT

%left MINUS PLUS
%left MUL DIV
```

```
%type <exp> exp fndef

%type <ival> number complex arg_num before_h hundreds
%type <name> fname;
%expect 0

%%

input: /* empty */
  | input func
  | error ;

func:  fndef IS exp ';' { build_function ($1, $3); } ;

fndef: arg_num FUNCTION fname { $$ = build_function_decl ( $3,
$1); } ;

arg_num: ARG_NUMBER    { printf("Arg number: %d\n",$1); $$ =
$1; };

fname: NAME            { printf("Func name: %s\n",$1); $$ =
$1; };

exp:  number      { $$ = build_int_2 ($1, $1 >= 0 ? 0 : -1); }
    | PARAM            { $$ = get_arg_decl ($1); }
    | exp PLUS exp {$$ = build (PLUS_EXPR, integer_type_node,
$1, $3);}
    | exp MINUS exp
            {
            $$ = build (MINUS_EXPR, integer_type_node, $1,
$3);
            }
   | exp MUL exp  { $$ = build (MULT_EXPR, integer_type_node,
$1, $3); }
   | exp DIV exp {$$=build (TRUNC_DIV_EXPR, integer_type_node,
$1,$3); }
     | FIRST exp THEN  { $$ = $2; }
     | '(' exp ')'     { $$ = $2; }
     | error           { $$ = error_mark_node; } ;

number:                { $$ = 0; }
  |   ZERO_NUMBER  { $$ = $1; }
  |   before_h          { $$ = $1; }
  |   hundreds          { $$ = $1; };

before_h:  SIMPLE_NUMBER              { $$ = $1; }

  |   TENS_NUMBER                { $$ = $1; }
  |   complex                    { $$ = $1; };
```

```
complex: COMPOSITE_NUMBER                { $$ = $1; }
  | COMPOSITE_NUMBER SIMPLE_NUMBER       { $$ = $1 + $2; };

hundreds: before_h HUNDRED               { $$ = $1 * 100; }
  | before_h HUNDRED AND before_h        { $$ = $1 * 100 + $4; }
  | before_h HUNDRED before_h            { $$ = $1 * 100 + $3; };
%%
```

Лабораторный практикум

К данному методическому пособию прилагаются материалы для проведения лабораторного проактикума по следующим разделам:

- разработка нового компилятора переднего плана (front end’a);
- примеры для изучения работы оптимизатора;
- пример описания архитектуры.

Исходные коды примеров и рекомендации доступны на сайте www.roman.nnov.ru

Рекомендуемая литература

1. А. Ахо, Р. Сети, Дж. Ульман, «Компиляторы: принципы, технологии и инструменты», М., «Вильямс», 2001.
2. Воеводин В. В. Отображение проблем вычислительной математики на архитектуру вычислительных систем. // Вычислительные методы и программирование, 2000.– Т. 1.– с. 73 - 44.
3. Воеводин Вл.В., Капитонова А.П. Методы описания и классификации вычислительных систем.–М.: Издательство МГУ.– 1994.– 380 с.
4. Евстигнеев В. А. Некоторые особенности программного обеспечения ЭВМ с длинным командным словом. // Программирование. – 1991.– №2.– с.69-80.
5. Евстигнеев В. А. Применение теории графов в программировании. // Под ред. А. П. Ершова.– М.:Наука. Гл. ред. физ.-мат- лит., 1985.– 352 с.
6. Евстигнеев В. А., Касьянов В. Н. Оптимизирующие преобразования в распараллеливающих компиляторах. // Программирование.– 1996.– № 6.– с. 12-26.
7. Евстигнеев В. А., Касьянов В. Н. Сводимые графы и граф-модели в программировании.– Новосибирск: Издательство ИДМИ, 1999.– 288 с.
8. Евстигнеев В. А., Серебряков В. А. Методы межпроцедурного анализа (обзор). // Программирование, 1992.– № 3.– с. 4-15.
9. Ершов А. П. Введение в теоретическое программирование (беседы о методе).– М.: Гл. ред. физ.-мат. лит. изд-ва "Наука".– 1977.– 288 с.
10. Касьянов В. Н. Оптимизирующие преобразования программ.– М. – Наука. Гл. ред. физ.-мат. лит., 1988.– 366 с.

11. Касьянов В. Н. Средства поддержки применения графов в программировании. // Проблемы программирования.– 2000.– №1-2.– с. 286-300.

12. Касьянов В. Н., Поттосин И. В. Методы построения трансляторов. – Новосибирск.– Наука, 1986.– 344 с.

13. Кузнецов О.П., Адельсон-Вельский Г.М. Дискретная математика для инженера.– М.: Энергия, 1980.– 344 с.

14. Т. Пратт, М. Зелковиц, «Языки программирования: разработка и реализация», СПб.: Питер, 2002

15. Скворцов С. В. Оптимизация кода для суперскалярных процессоров с использованием дизъюнктивных графов. // Программирование.– 1996, № 2.– с. 41-52.

16. Французов Ю. А. Обзор методов распараллеливания кода и программной конвейеризации. // Программирование.– 1992.– №3.– с. 16-30.

17. Французов Ю. А. Планирование потока команд с отложенным распределением регистров. // Программирование.– 1991, № 1.– с. 58-66.

18. Фути К., Судзуки Н. Языки программирования и схемотехника БИС. – М.: Мир.– 1988.– 224 с.

19. Шпаковский Г. И. Метод планирования трасс и архитектура ЭВМ со сверхдлинной командой // ЗРЭ.– 1991.– N 11.– с. 10-27.

20. Шпаковский Г. И. Организация параллельных ЭВМ и суперскалярных процессоров. – Мн.: Белгосуниверситет, 1996. – 284 с., ил.

21. Allan V., Jones R., Lee R., Allan S. Software Pipelining. // ACM Computing Surveys.– vol. 27, no. 3.– September 1995.– 90 p.

22. Aho A. V., Sethi R., Ullman J. D. Compilers: Principles, Techniques and Tools.– Reading, Mass: Addison-Wesley.– 500 p.– 1986. ISBN 0-201-10088-6.

23. Bacon D. F., Graham S. L., Sharp O. J. Compiler transformations for high-performance computing // ACM Computing Surveys.– 1994. V. 26. № 4.– PP. 345-420.

24. Bala V., Rubin N. Efficient Instruction Scheduling Using Finite State Automata. // In Proc. of IEEE Micro-28, 1995. P. 46-56.

25. Bashford S. Code Generation Techniques for Irregular Architectures // Tech. Rep. 596, Universitat Dortmund.– November 1995.– 120 p.

26. Beaty S. List Scheduling: Alone, with Foresight, and with Lookahead. // In Conf. on Massively Parallel Computing System: the Challenges of General-Purpose and Special Purpose Computing. – Ischia, Italy.– May 1994.– p. 246-253.

27. Briggs P. Register Allocation via Graph Coloring. // PhD thesis. Rice University, Houston, Texas.– April 1992.– 143 p.

28. Case B. Philips Hope to Displace DSPs with VLIW. // Microprocessor Report, 8(16).– 5 Dec. 1994.– p. 12-15.

29. Chen G., Smith M. Global Instruction Scheduling on Machine SUIF // In Workshop on Interaction between Compilers and Computer Architecture, High Performance Computer Architecture.– N 3.– Feb. 1997.– p. 176-187.

30. Ebcioglu K., Altman E. R. DAISY: Dynamic Compilation for 100% Architectural Compatibility. // In Proc. of 24th Intern. Symposium on Computer Architecture (ISCA).– June, 1997.– p. 26–37.

31. Ebcioglu K., Nakatani T. A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture. // In

Languages and Compilers for Parallel Computer.– MIT Press, Cambridge, MA, 1990.– p. 213-229.

32. Fauth A., Praet J. V., Freericks M. Describing instruction set processors using nML. // In Proc. of the European Design and Test Conference.– Paris.– March 1995.– p. 503-507.

33. Finkel R. Advanced Programming Languages Design. // Addison-Wesley, Kentucky.– 1996.– 372 p. ISBN 0-8053-1191-2.

34. Franke B., O'Boyle M. An empirical evaluation of high-level transformations for embedded processors. // In. Proc. of Int. Conference on Compilers and Architecture and Synthesis for Embedded System.– 2001.– p. 59-66.

35. Fraser C. W., Hanson D. R. A Retargetable Compiler for C: Design and Implementation. – Addison-Wesley.– Menlo Park, CA.– 1995.– 360 p.– ISBN 0-8053-1670-1.

36. Fraser C. W., Hanson D. R., Proebsting T. A. Engineering a simple, efficient code generator generator. // ACM Letters on Programming Languages and Systems.– Vol. 1.– 1992.– p. 213–226.

37. GCC Internals (www.gcc.gnu.org)

38. Hans-Peter Nilsson, “Porting GCC for Dunces”, 2000

39. Hanono S., Devadas S. Instruction Selection, Resource Allocation, and Scheduling in the Aviv Retargetable Code Generator.– 35th Design Automation Conference (DAC).– 1998.– p. 510-515.

40. Hwu W., et. al. Compiler Technology for Future Microprocessors. // In Proc. of IEEE.– Vol 83, No. 12.– Dec. 1995.– p. 1625-1640.

41. Intel Corp. Intel Architecture Optimization Manual. Order Number 242816-003, Intel.– 1997.– 240 p.

42. Johnson S. C. YACC – Yet Another Compiler-Compiler. // Computer Science Technical Report 32.– Bell Telephone Labs.– 1975. – 96 p.

43. Liao S., Devadas S., Keutzer K., Tjiang S., Wang A., Araujo G., Sudarsanam A., Malik S., Zivojnovic V., Meyr H. Code Generation and Optimization Techniques for Embedded Digital Signal Processors. // In Hardware/Software Co-Design, Kluwer Acad. Pub.– 1995.– pp. 599-604.

44. Makarov V. N., The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC, GCC Developers Summit 2003

45. Patterson D. A., Hennessy J. L. Computer Organization and Design: The Hard-ware/Software Interface.– Morgan Kaufmann Publishers.– Second Edition, 1998.– 993 p.

46. Poletto M., Sarkar V. Linear Scan Register Allocation. ACM TOPLAS, 21(5).– 1999.– pp.895-913.

47. Proceedings of the GCC Developers Summit, Ottawa, Ontario 2003

48. Sreejith K Menon, “GCC Frontend howto”, 2002

49. Stallman R. Using and Porting the GNU Compiler Collection.– Addison-Vesley Publishing, New York.– 2000.– 556 p.

50. Steven S. Muchnick, “Advanced Compiler Design and Implementation”, Academic press, London, 1997

51. Wilson T. et al. An ILP-Based Approach to Code Generation. // In Code Genera-tion for Embedded Processors.– Kluwer Academic Publishers, 1995.– pp. 103-118.