Задачи компиляции для современных архитектур

Андрей Белеванцев <abel@ispras.ru>

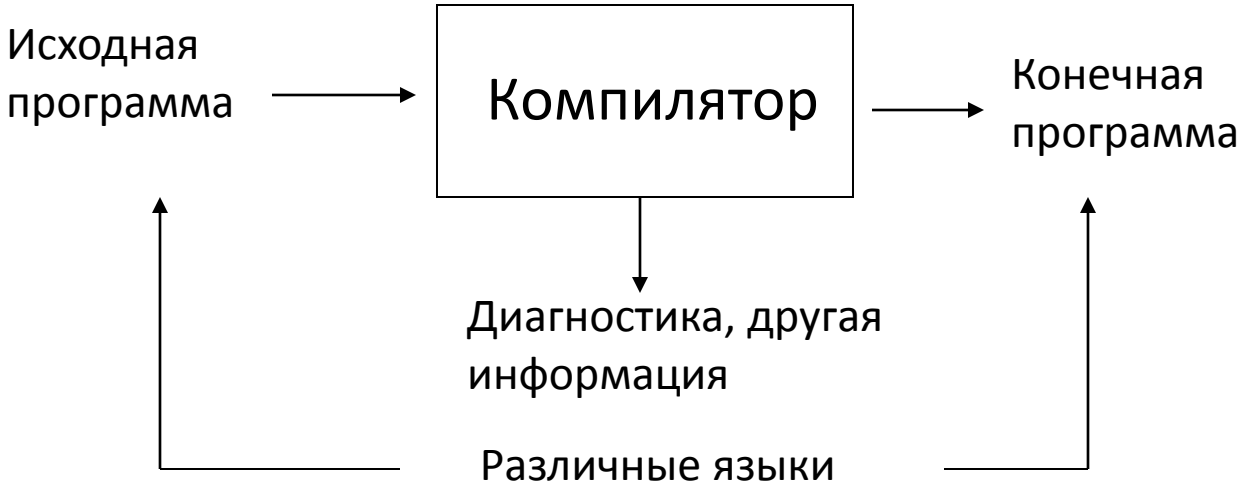
Отдел компиляторных технологий

Институт системного программирования РАН

О чем эта презентация

- Что такое компилятор
- Как устроен оптимизирующий компилятор
- Компилятор GCC
- Пример: планирование команд
- Пример: компиляция с учетом свойств аппаратуры и поведения пользователя
- Пример: статический анализ исходного кода
- Что нужно знать для работы над компилятором

Компиляторы

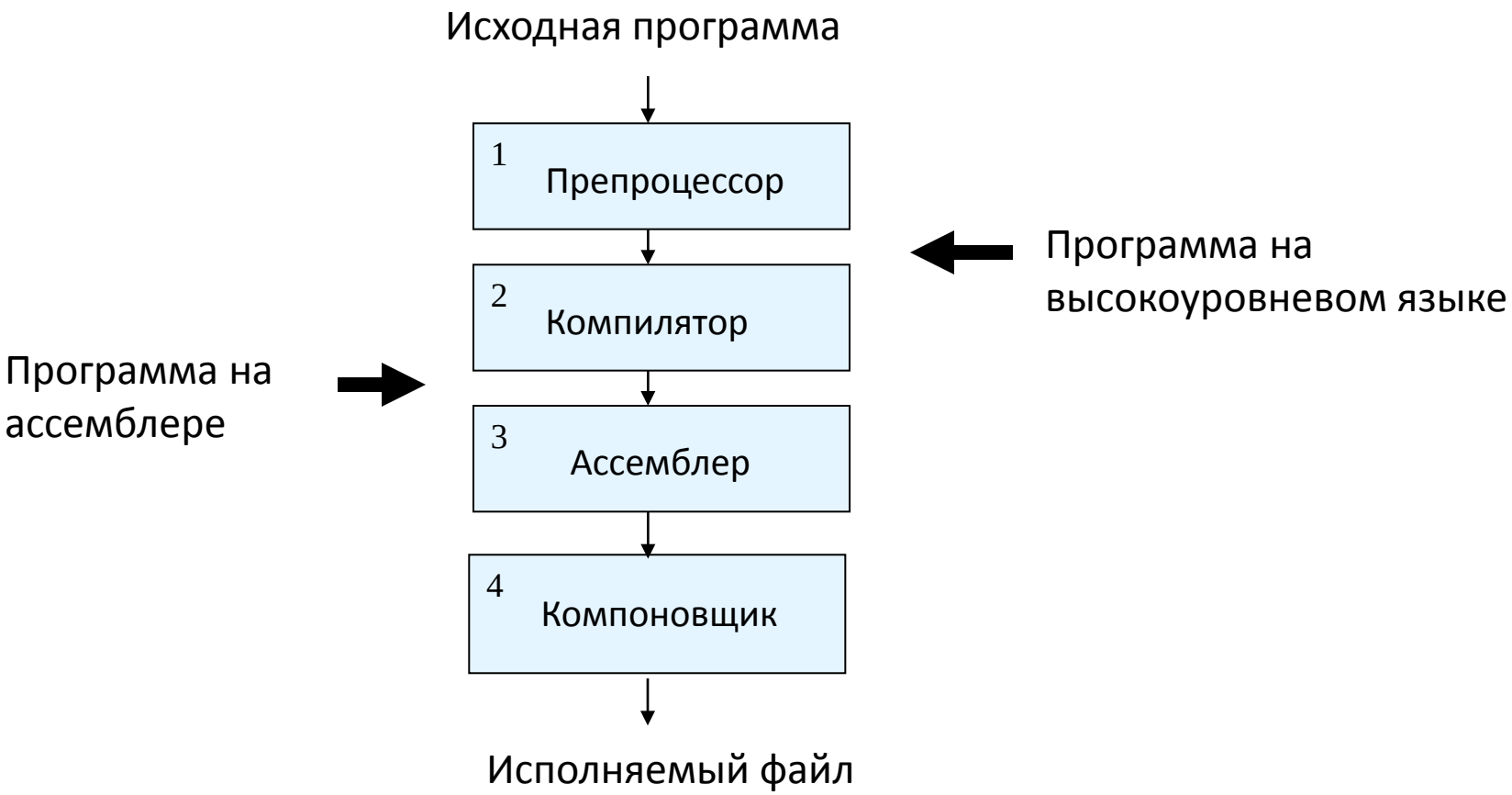


- GCC: C, C++, Fortran, Java->ассемблер целевой машины
- LaTeX: текст с макрокомандами-> .DVI файл
- Компиляторы VHDL: VHDL->схема чипа
- Базы данных: SQL->план выполнения запроса
- PROMT: английский->русский

Оптимизирующие компиляторы

- По программе на исходном языке можно построить множество семантически эквивалентных на целевом языке
- Хотим получить конечную программу, оптимальную по некоторому критерию
  - Скорость выполнения
  - Размер кода
  - Энергопотребление
  - Размеры “пустых” мест на сверстанной странице (TeX)
- Задача построения оптимальной программы обычно алгоритмически неразрешима

Схема компиляции



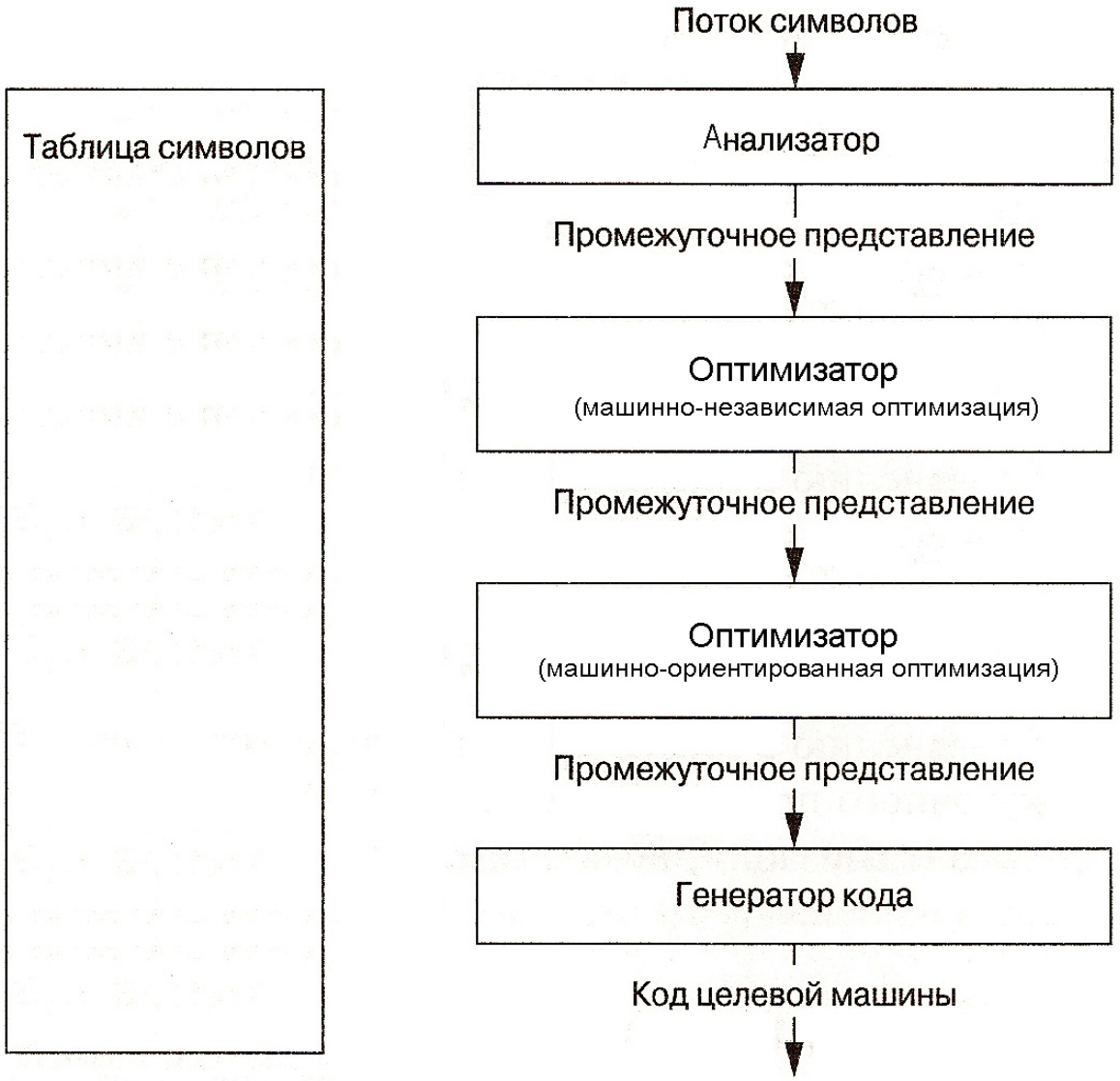
Что ещё делает компилятор

- Диагностика (ошибки и предупреждения)
  - test.c:10: error: 'N' undeclared (first use in this function)
- Выдача отладочной информации
  - gcc –g –o test test.c
  - gdb test
- Выдача “другой” информации (ход оптимизаций, ...)
- Соответствие целевой платформе
  - Память под глобальные и статические данные
  - Подготовка информации для ассемблера и компоновщика
  - ABI (вызовы функций, выравнивание)
  - ...

Модель компиляции

- Два этапа – анализ и синтез
- Анализ: разбор исходной программы, определение выполняемых операций, выдача диагностики, генерация *внутреннего представления* (дерево операций, трехадресный код)
- Синтез: генерация эквивалентной программы на целевом языке по внутреннему представлению

Структура оптимизирующего компилятора



Препроцессирование

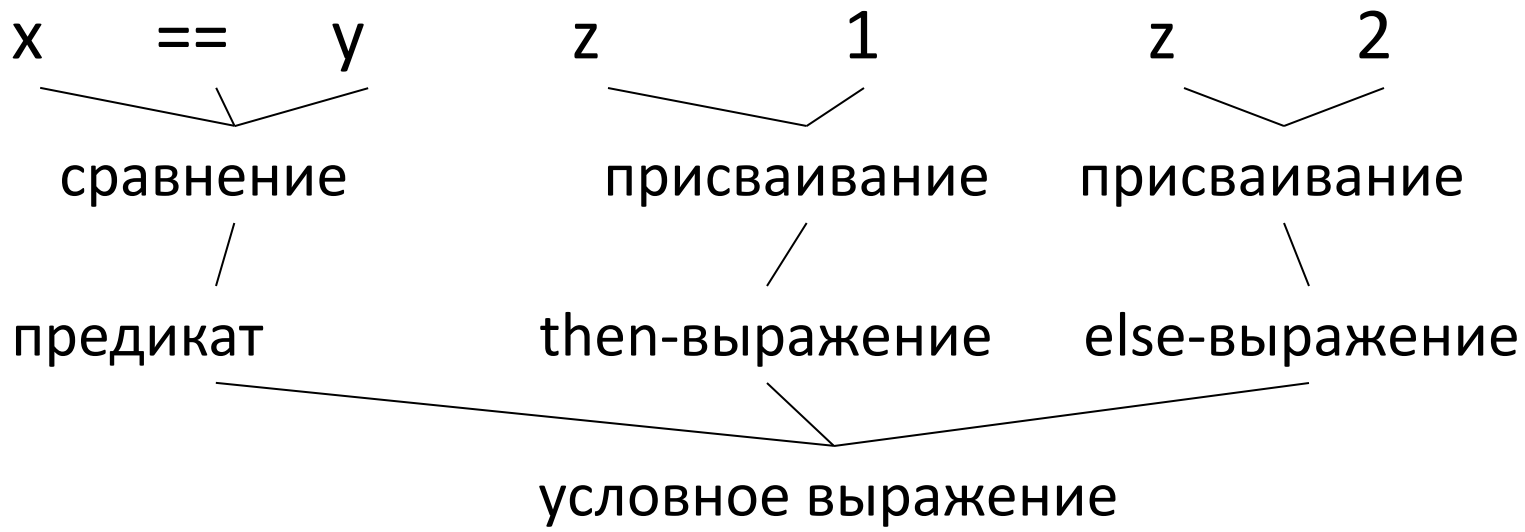
- Сформировать входную программу для компилятора
- Макросы – текстовые подстановки
  - #define i j // Happy debugging!
- Включение файлов
  - #include “headers.h”
- Скрытие деталей реализации
  - # define tolower(c) __tobody (c, tolower, *__ctype_tolower_loc (), (c))

Лексический анализ

- Разбить текст (программу) на слова
- Выполняется за один проход по тексту
- Незначащие символы удаляются
  - Пробелы, комментарии
- Мама мыла раму, а папа чинил машину.
  - “Мама”, “мыла”, “раму”, “,”, “а”, “папа”, “чинил”, “машину”, “.”
- if x == y then z = 1; else z = 2;
  - if, x, ==, y, then, z, =, 1, ;, else, z, =, 2, ;

Синтаксический анализ

- Выделить предложения и разобрать их структуру по правилам грамматики языка
- Часто требует рекурсивного обхода дерева, заглядывания вперед на несколько шагов
- if x == y then z = 1; else z = 2;



Семантический анализ

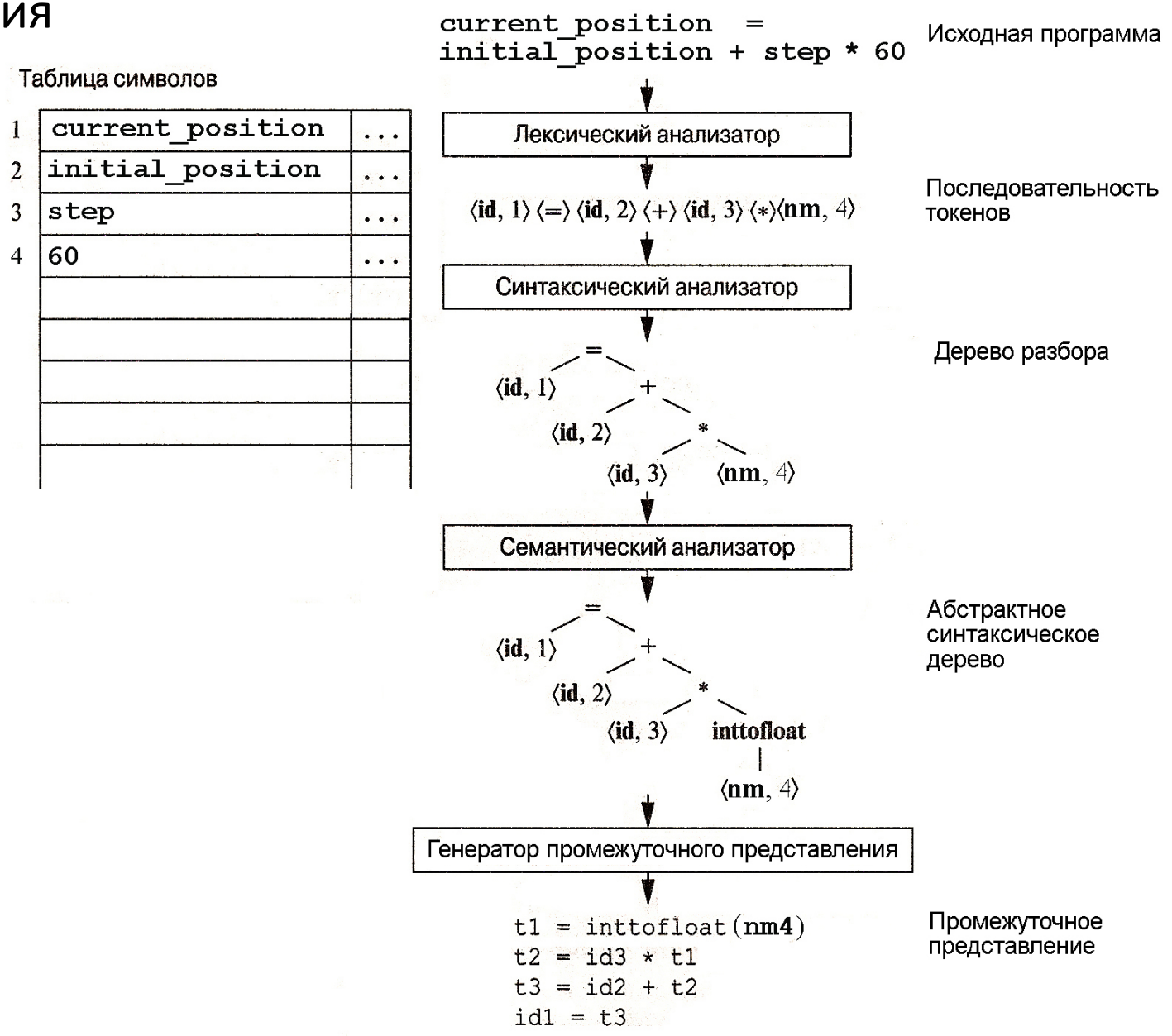
- Предложения языка могут быть многозначными
- Игорь сказал, что Сергей забыл его учебник в общаге
  - Чей учебник забыл Сергей?
- Jack said Jack forgot his textbook at home
  - Сколько всего Джеков?
  - Кто из них забыл учебник?

Семантический анализ

- Грамматика языка дополняется правилами, позволяющими избегать двусмысленных трактовок
- Внутреннее определение переменной Jack перекрывает внешнее
- Необходимы дополнительные проверки этих правил
  - Не больше одного Джека на уровень вложенности
- “Лена забыл дома своё учебник”
  - Несоответствие типов между “Лена” и “забыл”, “учебник” и “своё”

```
{
    int Jack = 3;
    {
        int Jack = 4;
        System.out.
            print(Jack);
    }
}
```

Front-end (передний план): Генерация промежуточного представления



Оптимизация программы

- Наиболее важная и “наукоемкая” часть компилятора (90%)
- Оптимизация обычно состоит из двух частей:
  - Анализ программы – определение необходимых свойств
  - Преобразование программы – поиск выгодных упрощений и их применение
- Машинно-независимые оптимизации
  - Имеют смысл для любой платформы
- Машинно-зависимые оптимизации
  - Во внутреннее представление привносится знание о целевой архитектуре – адресная арифметика, регистры
- Оптимизации, использующие профиль программы, т.е. знание о её типичном поведении
  - Двухпроходная компиляция
- Оптимизации во время выполнения программы

Пример программы в промежуточном представлении (программа построения единичной матрицы)

◊ ***Исходная программа на языке С***

```
for (i = 1; i = 10; i++)
     for (j = 1; j = 10; j++)
             a[i][j] = 0.0;
for (i = 1; i = 10; i++)
     a[i][i] = 1.0;
```

◊ ***Замечания***.

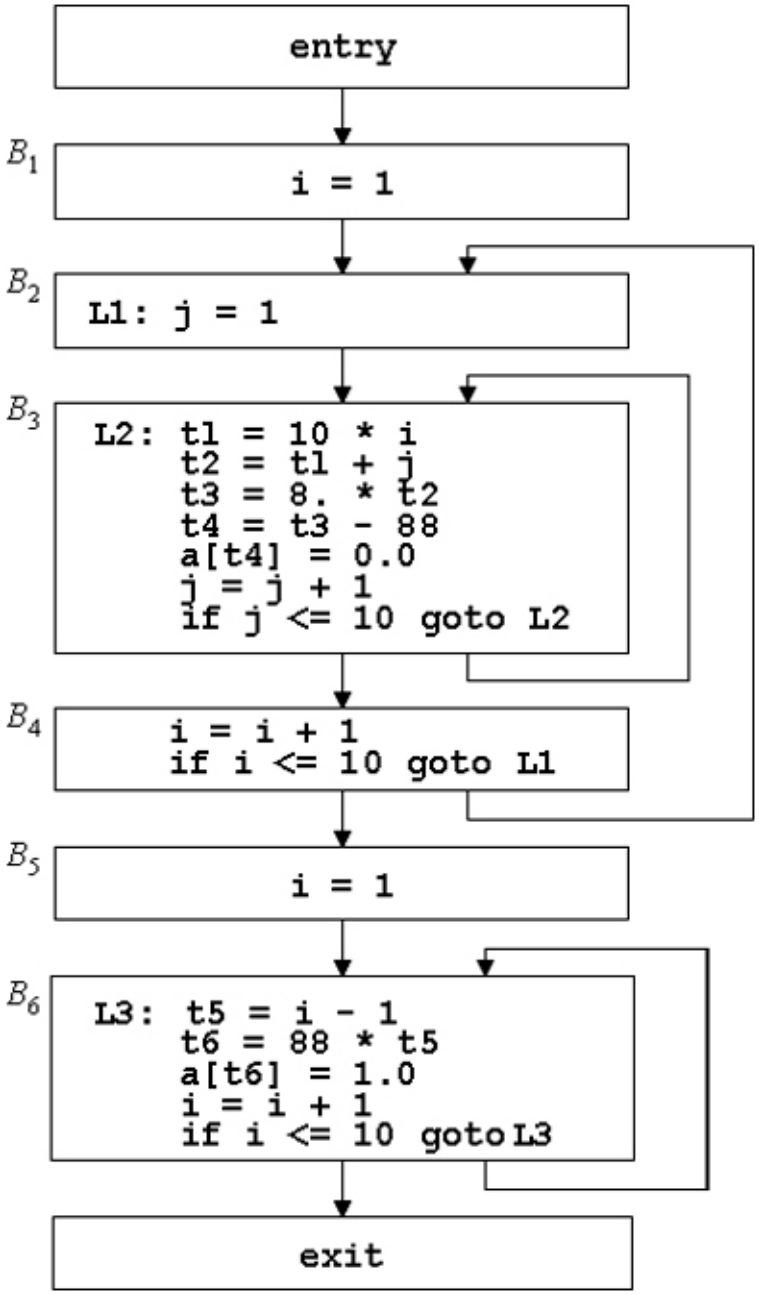
- Массив **a[10][10]** хранится построчно**.**
- каждый элемент массива является числом с плавающей точкой и имеет размер 8 байт.

◊ ***Ее промежуточное представление***:

```
( 1) i = 1
( 2) j = 1
( 3) t1 = 10 * i
( 4) t2 = tl + j
( 5) t3 = 8. * t2
( 6) t4 = t3 - 88
( 7) a[t4] = 0.0
( 8) j = j + 1
( 9) if j <= 10 goto(3)
(10) i = i + 1
(11) if i <= 10 goto(2)
(12) i = 1
(13) t5 = i - 1
(14) t6 = 88 * t5
(15) a[t6] = 1.0
(16) i = i + 1
(17) if i <= 10 goto(13)
```

Граф потока управления

На рисунке – построенный граф потока. Для удобства анализа к графу потока добавляют два дополнительных узла: *entry* (вход) и *exit* (выход).



Локальная оптимизация

◊ Базовый блок задается тройкой объектов: $B = \langle P, Input, Output \rangle$

P – последовательность инструкций

$Input$ – множество переменных, вычисляемых до входа в и используемых в блоке B

$Output$ – множество переменных, вычисляемых в блоке B и используемых в других блоках

◊ Локальная оптимизация – это выполнение следующих преобразований внутри базового блока:

- Удаление *общих подвыражений* (инструкций, которые повторно вычисляют уже вычисленные значения).
- Удаление *мертвого кода* (инструкций, вычисляющих значения, которые впоследствии не используются).
- *Сворачивание констант* (вычисление выражений, все операнды которых – константы с известными значениями).
- Изменение порядка инструкций, там, где это возможно, чтобы сократить время хранения временных значений на регистрах.

◊ *Метод нумерации значений* позволяет выполнить всю локальную оптимизацию за один просмотр базового блока

Недостаточность локальной оптимизации

В фрагменте программы *Quicksort*:

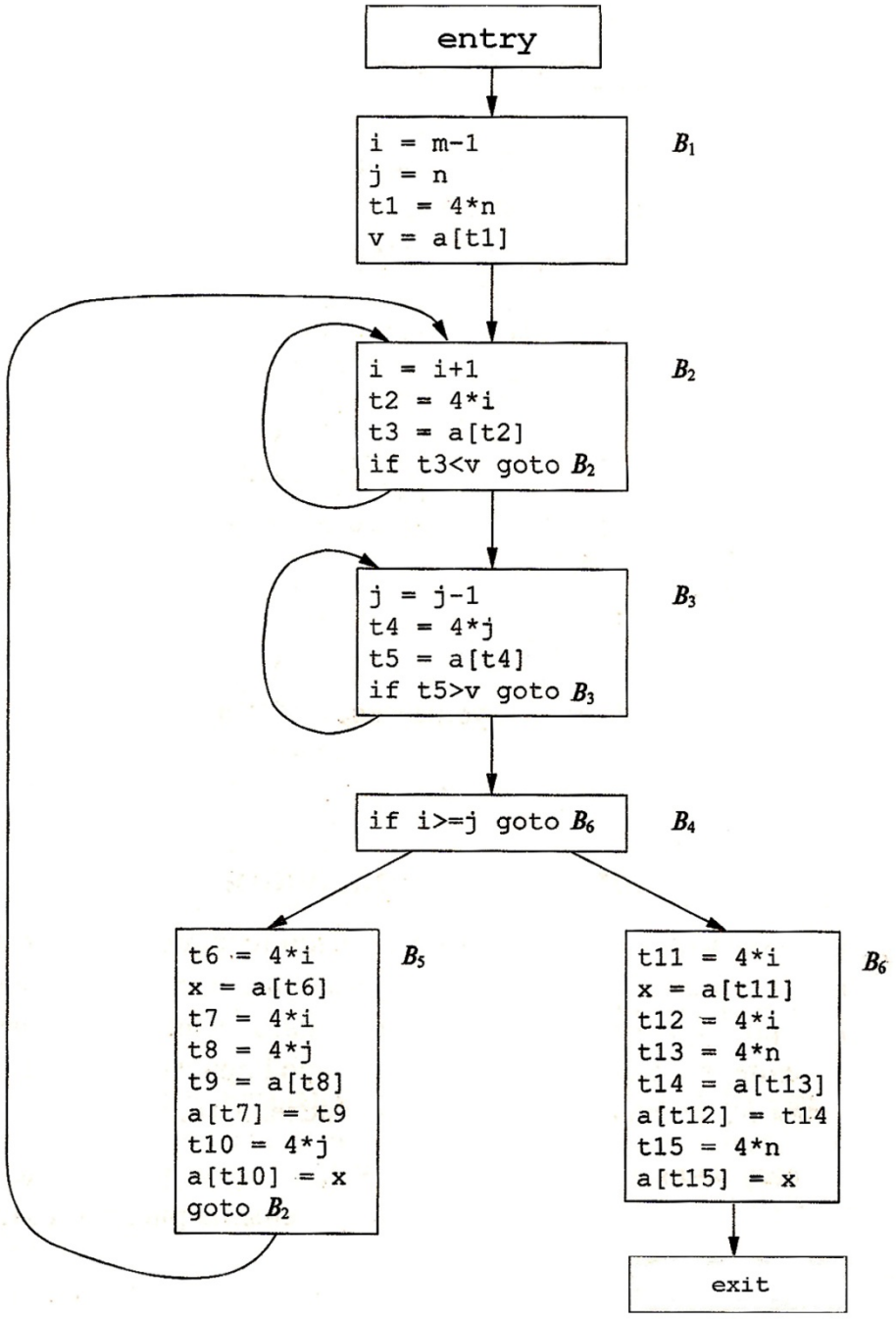
локальная оптимизация позволяет выявить и исключить лишние инструкции
t7 = 4*i
t10 = 4*j

Блок B_5 до оптимизации

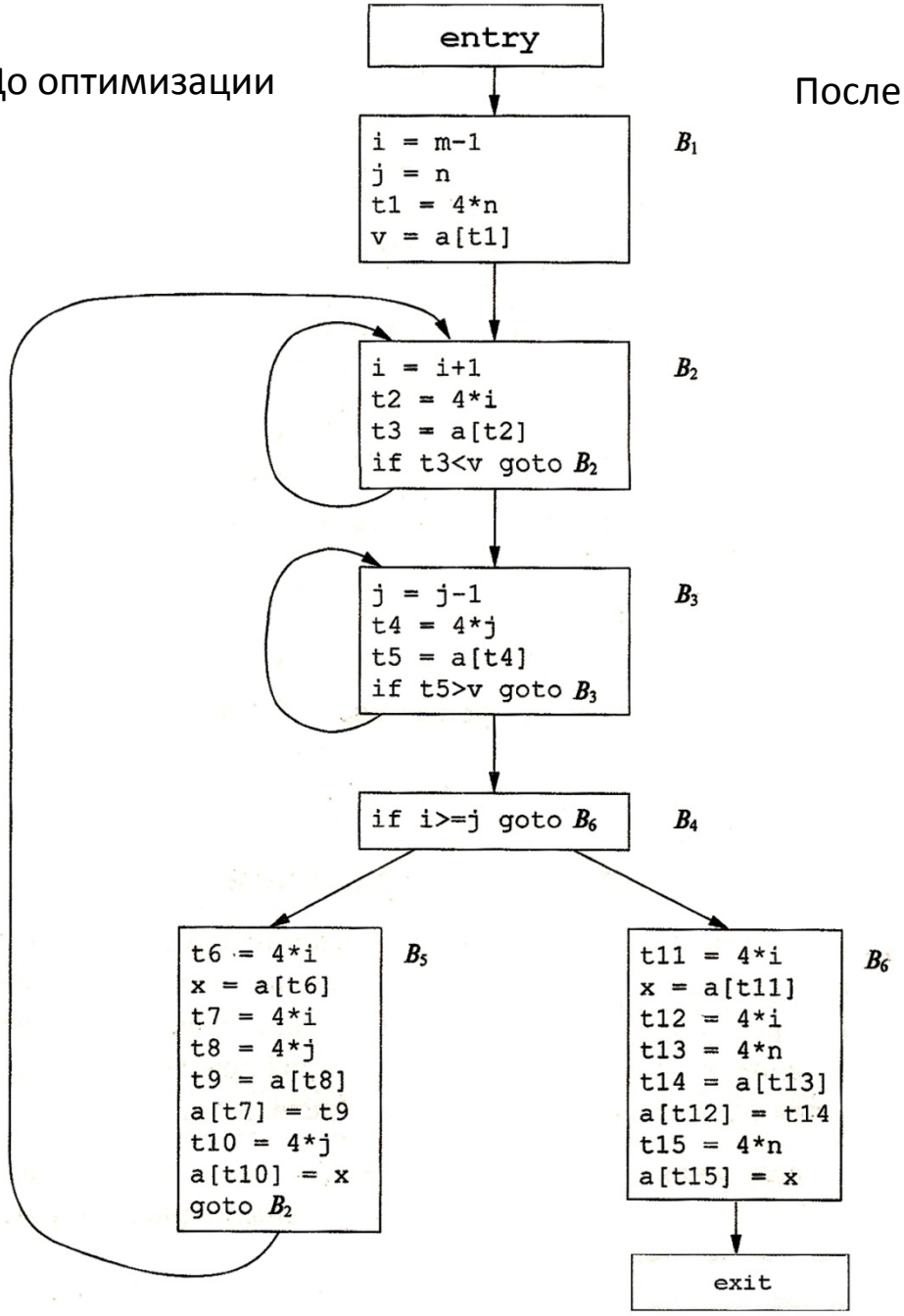
```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```
B_5

Блок B_5 после локальной оптимизации

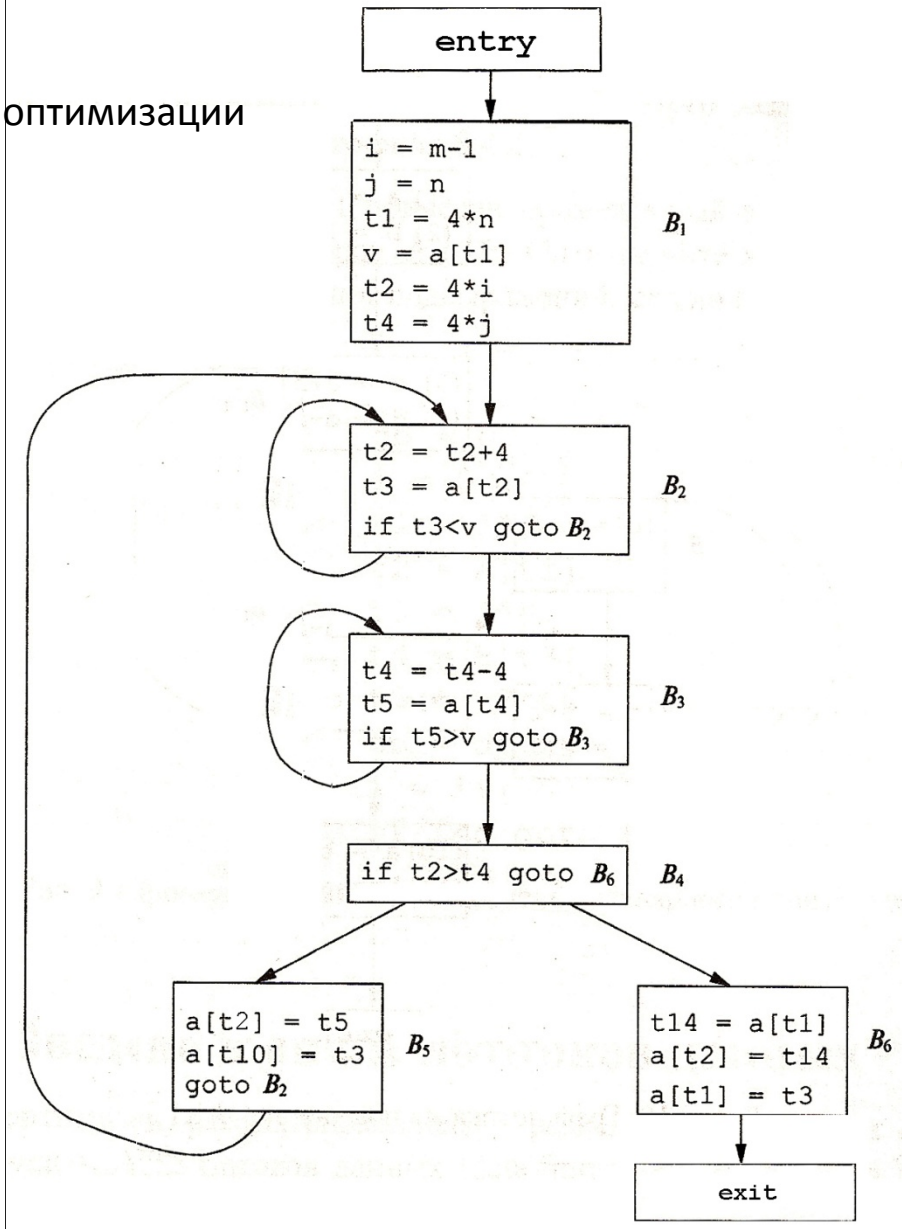
```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```
B_5



До оптимизации



После оптимизации



Глобальная оптимизация всего фрагмента *Quicksort*

Глобальная оптимизация

- Скалярные оптимизации:

  исключение избыточных вычислений (в том числе, частично избыточных, т.е. таких, которые избыточны только на части трасс программы),

  вынесение инвариантных вычислений из циклов,

  исключение мертвого кода (бесполезных вычислений)

  распространение значений

  распространение констант

- Оптимизации циклов (над массивами)
- Автоматическая векторизация
- Необходимые анализы (анализ алиасов, анализ цикловых зависимостей, интервальный анализ/нумерация значений и т.д.)

Кодогенерация

- Получить ассемблер целевой машины по внутреннему представлению
- Выбор инструкций

```
int x, double y;          mov r1, x; add r1, 20; mov y, r1;
y =  x + 20;        или   mov r1, x; mov r2, 20; add r1,r2; mov y,r1;
```

- Распределение регистров
  - Количество физических регистров машины меньше, чем переменных в программе
  - Для вычислений необходимо использовать промежуточную память
- Поддержка ABI целевой платформы

Машинно-зависимые оптимизации

- Внутреннее представление становится машинно-зависимым
  - Адресная арифметика, указатель на стек, регистры

int a[]; a[i] = a[j]*3 + a[k]; => a1 = a + j*sizeof(int); t1 = *a1; t1 = t1 * 3;
a2 = a + k*sizeof(int); t2=*a2; t1 = t1+t2; a3 = a + i*sizeof(int); *a3 = t2;

- Распределение регистров

r5 = r4 + r8*4; r1 = [r5]; r1 = r1*3; r6 = r4 + r9*4; r2=[r6]; r1 = r1+r2;
r7 = r4 + r10*4; [r7]=r2;

- Планирование команд

r5 = r4 + r8*4; r6 = r4 + r9*4; r7 = r4 + r10*4; r1 = [r5]; r2=[r6]; r1 = r1*3;

r1 = r1+r2; [r7]=r2;

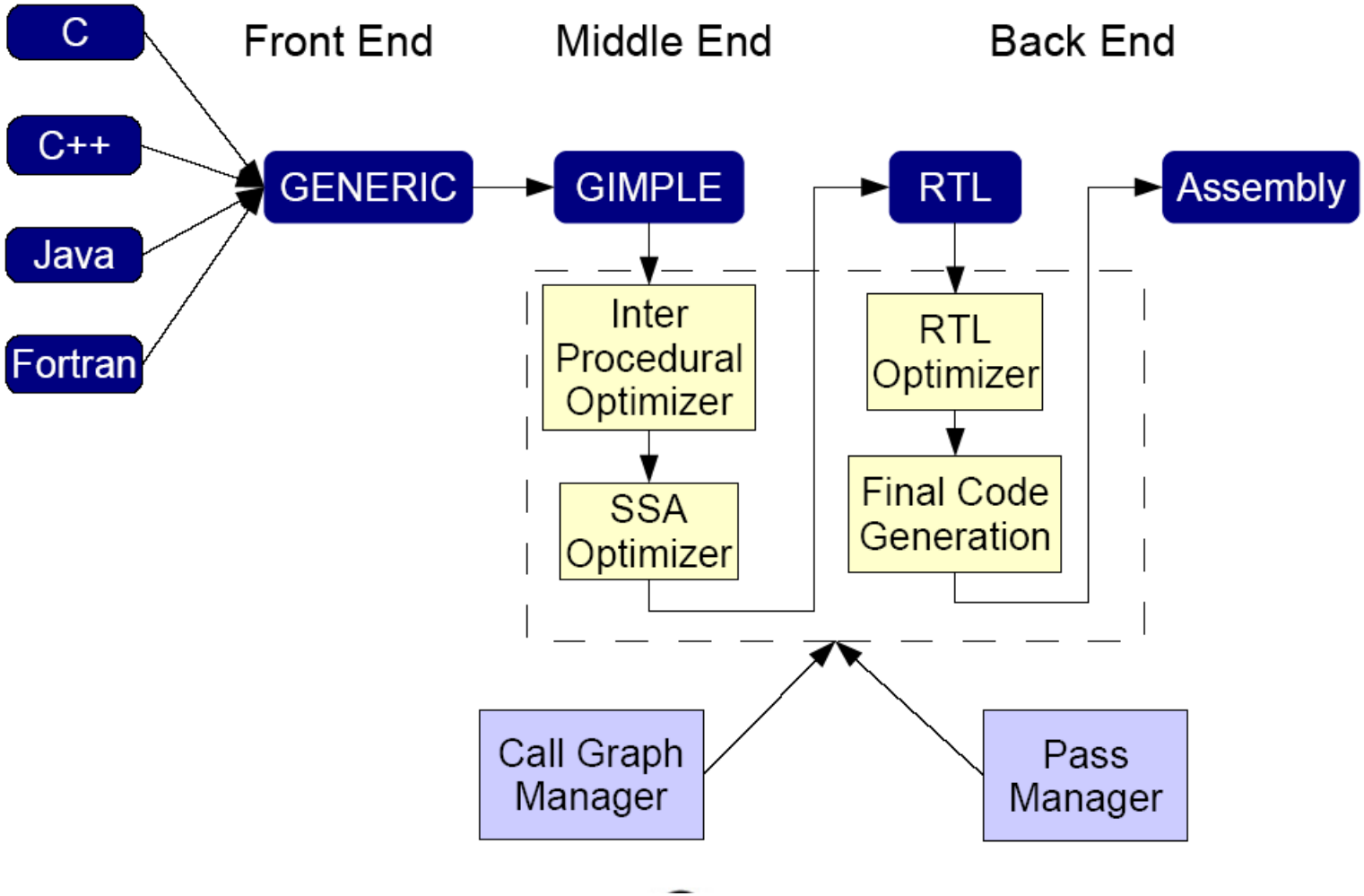
GCC (GNU Compiler Collection)

- Последний релиз: 4.7.0 (22 марта 2012)
- GCC 25 лет!
- Поддержка языков: Ada (GNAT), C (GCC), C++ (G++), Fortran (GFortran), Java (GCJ), Objective-C, Objective-C++
  - Неофициальные релизы: Pascal, Modula-2, VHDL, PL/I
- 43 целевые платформы, поддерживающих разные варианты процессоров
  - Множество неофициальных портов
- 150+ анализов и трансформаций

GCC- II

- Есть на всех популярных платформах, от встраиваемых (arm/avr) до мейнфреймов (z310)
- Стандартный компилятор для UNIX/Linux
- Лицензия GPL v3, копирайт FSF
- Высокое качество кода
  - “Самосборка” (bootstrap) без “предупреждений”
  - Большой набор тестов
- Используется в индустрии и в исследованиях
  - GCC 4.6: 550K ядро, 350K кодогенераторы, 800K языки, 1600K поддержка времени выполнения, 800K тесты, всего >4.2MLoc
  - “Оценивается ” sloccount на 1300 человеко-лет

Архитектура компилятора GCC



Внутреннее представление GIMPLE

```
struct A { A(); ~A(); };

  int i;
  int g();
  void f()
  {
    A a;
    int j = (--i, i ? 0 : 1);

    for (int x = 42; x > 0; --x)
      {
        i += g()*4 + 32;
      }
  }
```

```
void f()
  {
    int i.0;
    int T.1;
    int iftmp.2;
    int T.3;
    int T.4;
    int T.5;
    int T.6;

    {
      struct A a;
      int j;

      __comp_ctor (&a);
      try
        {
          i.0 = i;
          T.1 = i.0 - 1;
          i = T.1;
          i.0 = i;
```

```
if (i.0 == 0)
  iftmp.2 = 1;
else
  iftmp.2 = 0;
j = iftmp.2;
{
  int x;

  x = 42;
  goto test;
  loop:;

  T.3 = g ();
  T.4 = T.3 * 4;
  i.0 = i;
  T.5 = T.4 + i.0;
  T.6 = T.5 + 32;
  i = T.6;
  x = x - 1;
```

```
test:;
      if (x > 0)
        goto loop;
      else
        goto break_;
      break_:;
      }
  }
finally
  {
    __comp_dtor (&a);
  }
}
```

Пример: планирование команд

- Современные процессоры обладают параллельно работающими конвейерными функциональными устройствами
  - При полной загрузке – несколько (6-9) команд за такт
  - Устройства – целочисленные, FPU, переходы, работа с памятью
- Планирование: как выбрать последовательность команд, чтобы все устройства были загружены, все данные готовы в нужный момент?
- Как быстро может выполняться программа?
  - Доступный параллелизм процессора.
  - Потенциальный параллелизм программы.
  - Возможность выделить параллелизм в исходной программе.
  - Наша способность спланировать наилучшее параллельное выполнение при заданных ограничениях планирования.

Сохранение семантики программы

- Ограничения на порядок команд
  - **Ограничения управления.** Все операции, выполняемые в исходной программе, должны выполняться и в оптимизированной программе при тех же условиях (зависимости по управлению).
  - **Ограничения данных.** Операции в оптимизированной программе должны выдать те же результаты, что и соответствующие операции в исходной программе (зависимости по данным – порядок чтения и записей должен сохраняться)
  - **Ограничения ресурсов.** Планирование кода не должно требовать чрезмерного количества ресурсов машины (нельзя выдавать на одном такте три загрузки, если есть одно устройство).

Анализ зависимостей по данным

- Алгоритмически неразрешимая задача в общем случае
- Как узнать, что две данные команды обращаются к одной ячейке памяти?

```
a = 1
*p = 2
b = a
```

- Одна зависимость между 1 и 3?
  - Если знаем, что указатель **p** не может указывать на **a**!
- Если это не так, тогда возникают еще две зависимости:
  - истинная зависимость между командами 2 и 3
  - зависимость по записи между командами 1 и 2.
- Анализ алиасов (псевдонимов)
  - Межпроцедурный анализ (foo (&p) вместо *p = 2)

Модель ресурсов процессора

- Каждая операция имеет множества входных и выходных операндов и необходимые ресурсы для ее выполнения
- Входная задержка: когда нужно значение входных операндов относительно начала выполнения (обычно сразу)
- Выходная задержка: когда готовы значения выходных операндов относительно начала выполнения (обычно через некоторое фиксированное число тактов)
- Моделирование ресурсов – какие функциональные устройства нужны для выполнения операции и на каких тактах
- Более сложные ограничения (после выдачи операции X операция Y может быть выдана через один такт)

Граф зависимостей по данным

- Узлы – операции, ребра – зависимости по данным
- Каждое ребро помечено задержкой – через сколько тактов конец дуги может выполняться после начала
  - В GCC анализ зависимостей генерирует списки прямых и обратных зависимостей для инструкций
- Вершины могут помечаться таблицей резервирования ресурсов (см. модель процессора)
  - В GCC вместо этого по описанию модели процессора генерируется конечный автомат, отслеживающий состояние процессора
  - Интерфейсы автомата:
    - Через сколько тактов можно выдать данную операцию при данном состоянии?
    - Какова задержка между данной парой инструкций?

Постановка задачи планирования

- Рассмотрим область графа потока управления программы
  - Область должна быть ациклической (иначе граф зависимости по данным может содержать циклы)
  - В простом случае область = базовый блок, в более сложных – подграф без циклов с одним входом и несколькими выходами (способы поиска таких областей в графе потока управления известны)
- Построим зависимости по данным и по управлению для данной области
- Как можно переставить операции в области, чтобы:
  - Все зависимости были сохранены
  - Для данного процессора модельное время выполнения области было минимальным
- Все перестановки перебирать не хочется!

Планирование базового блока

- Самый простой случай – нет зависимостей по управлению
- Определим критический путь в графе зависимостей (самый длинный путь с учетом задержек)
  - Если ограничений по ресурсам нет, то длина критического пути есть длина оптимального расписания
- Эвристика – обходим граф зависимостей в топологическом порядке
  - Порядок гарантирует, что все операции-предки уже запланированы
  - Для очередной вершины определяем минимально возможный такт планирования, исходя из ограничений по ресурсам
- Как выбрать “наилучший” порядок?
  - Приоритеты вершин – высота в графе, использование критических ресурсов, исходный порядок

Планирование области

- Нужно обращать внимание на зависимости по управлению
- Пусть операция перемещается вверх по пути управления
  - Используется отношение доминирования
  - Эквивалентность по управлению (доминирование + постдоминирование) – перемещение безопасно
  - Целевой блок доминирует над исходным – безопасно в том случае, если нет операций с памятью и не нарушаются области жизни регистров
  - Целевой блок не доминирует над исходным – нужен компенсирующий код
- Выгодно ли перемещение? Желательно, чтобы ресурсы процессора в целевом блоке простаивали
- Как меняются зависимости по данным между операциями в разных блоках?
  - Операции, которые не могли принадлежать одному пути управления, после перемещения могут появиться на одном пути (новые зависимости)

Спекулятивное выполнение

- Операция выполняется до того, как будет ясно, что она нужна, либо до того, как готовы операнды
- По управлению:

Исходный код:

```
if (a>b) {
   load(ld_addr, target)

   use(target)
}
```

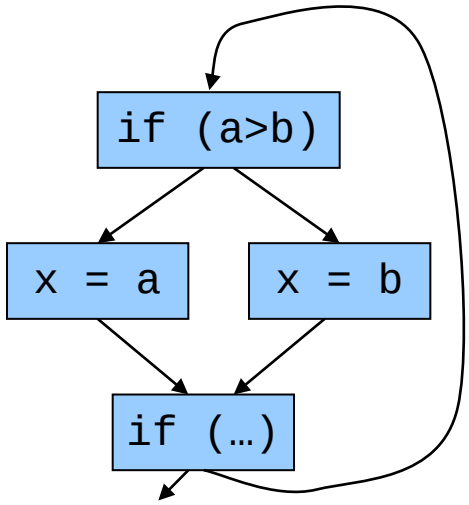
Преобразованный код:

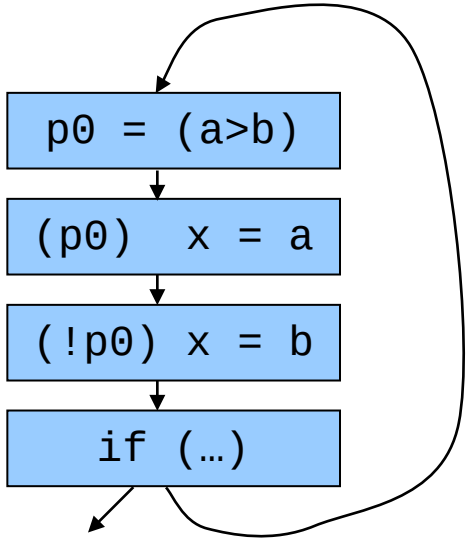
```
/* Начать выполнять загрузку раньше */
sload(ld_addr, target)

if (a>b) {
  /* Проверить, не вызвала ли она
   исключений */
  scheck(target, recovery)
  use(target)
}
```

Условное выполнение

- Команды условного перехода заменяются командами условного выполнения
- Количество ветвлений сокращается, это помогает при конвейеризации циклов





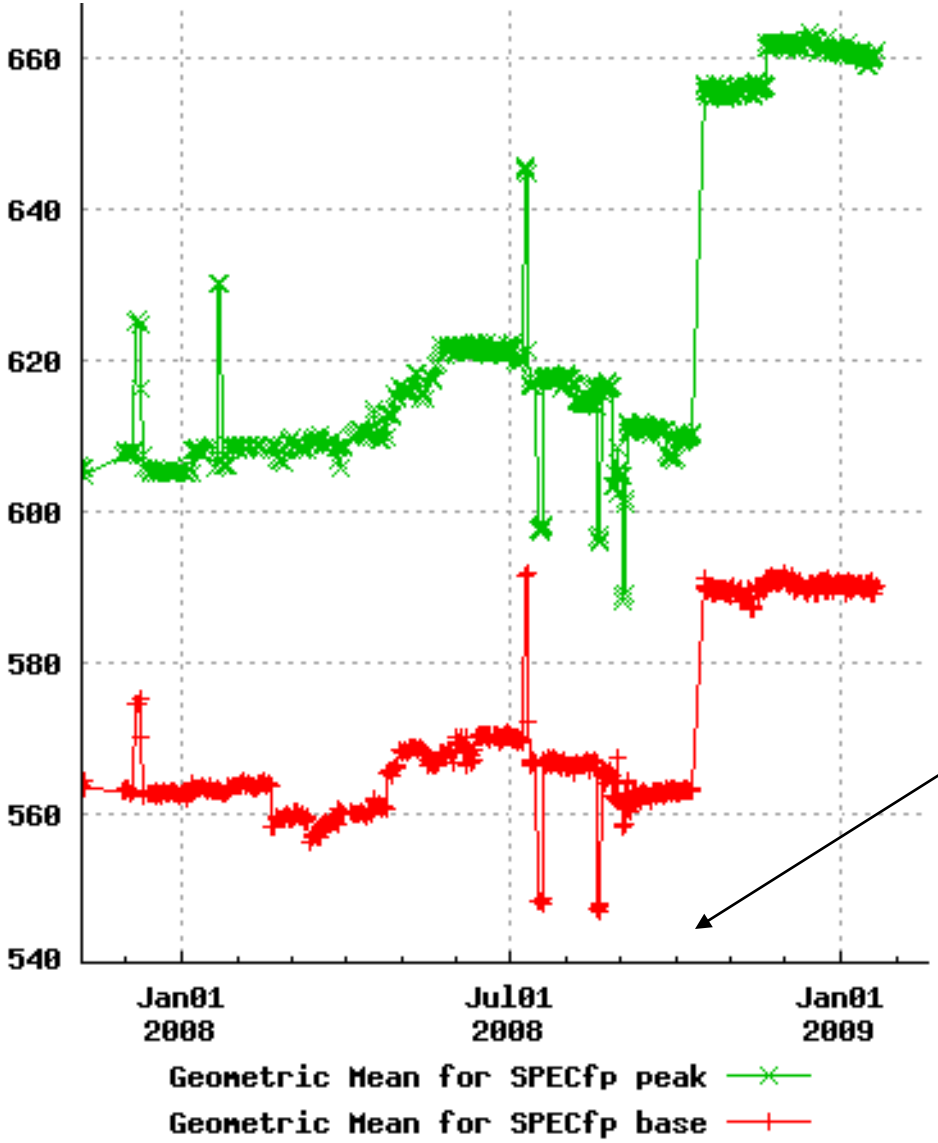
Планирование команд: итоги

- Важная оптимизация, позволяющая использовать мелкозернистый параллелизм процессора
- Привлекаются анализы:
  - Граф потока управления, дерево доминирования
  - Выделение областей, циклов
  - Зависимости по данным, анализ алиасов
  - Области жизни регистров (bitmaps)
  - Операции на графе зависимостей (критический путь)
  - Информация об операциях (исключения, спекулятивные версии)
  - Модель функциональных устройств процессора
  - Корректна ли операция для данной архитектуры

Пример нашей работы в GCC

- Новый планировщик с поддержкой конвейеризации
  - Поддерживает ряд преобразований команд (спекулятивное и условное выполнение, переименование регистров)
  - Поддерживает конвейеризацию циклов, в том числе вложенных, с неизвестным заранее числом итераций и с большим числом ветвлений
  - Размер патча более 800Кб относительно основной ветви
  - 15 месяцев разработки, 9 месяцев настройки производительности, поддержка кода
  - Включен в GCC 4.4
  - На наборе тестов SPEC FP 2000 получено ускорение в среднем 3.9% (или ~5% с учетом оптимизаций кодогенератора, включенных в GCC ранее), в отдельных тестах до 10%

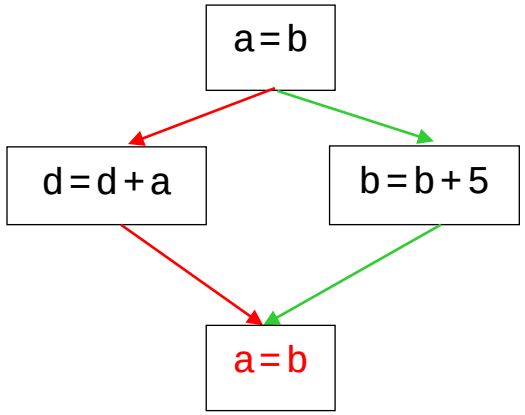
Результаты планировщика

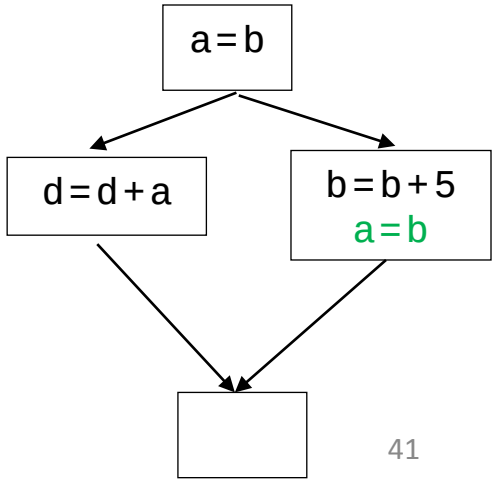


Включение нескольких новых оптимизаций, в том числе разработанного нами планировщика команд в основную ветвь GCC

Оптимизации, управляемые профилем

- Обычные оптимизации рассчитаны на “среднее” поведение программы
  - Например, обе ветки условного перехода выполняются с одинаковой вероятностью
- В реальности это верно не всегда
- Решение: собрать информацию о поведении программы на типичных входных данных (“профиль”) и использовать её
- Минус – усложняется процесс компиляции для пользователя





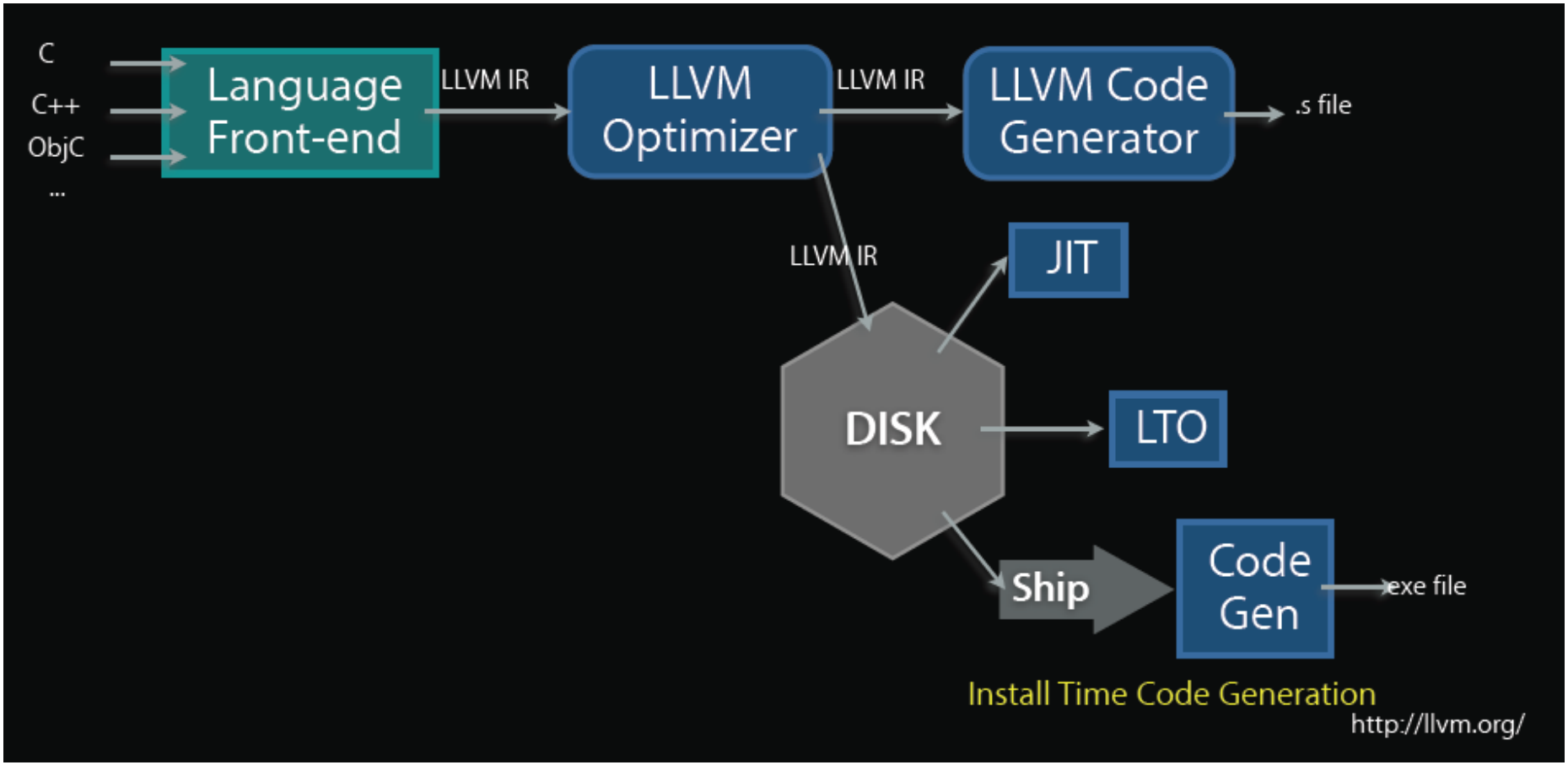
Динамическая оптимизация

- Скриптовые языки и языки с виртуальной машиной (Java, C#, Python) используют динамическую оптимизацию
  - При оптимизации учитывается профиль конкретного пользователя и конкретная архитектура → ускорение!
  - Виртуальная машина и распространение в байткоде позволяют экономить на развертывании
    - Одна версия программы для N архитектур
- Программы на языках общего назначения (C/C++) традиционно оптимизируются статически
  - Нет байткода → Много версий программы в бинарном коде (i386, x86-64, powerpc…)
  - Нет оптимизаций под конкретного пользователя/процессор
- Можно ли их тоже оптимизировать динамически?
  - Нужен машинно-независимый байт-код для C/C++
  - Распространяем в байт-коде → не делаем кодогенерацию на стороне разработчика → нужна динамическая оптимизация на стороне пользователя
    - Получаем учет профиля пользователя
    - Получаем учет архитектуры пользователя
  - Можем ли получить сравнимую производительность?

LLVM

- LLVM: открытая компиляторная инфраструктура (Apple)
  - Последний релиз: 3.0 (3.1 на подходе)
  - Поддержка Си-семейства языков (C/C++/ObjC-C++) популярных платформ (x86, arm, ppc, …)
  - JIT-компилятор, интерпретатор, отладчик и т.п.
- Основные положительные свойства
  - Модульная (набор библиотек)
    - Компилятор, интерпретатор, JIT, отладчик
  - Единое сериализуемое внутреннее представление
  - Лицензия BSD

Архитектура LLVM



Динамическая оптимизация с LLVM

- Для экспериментов необходима богатая компиляторная инфраструктура и виртуальная машина
  - Байткод
  - JIT-компилятор
  - Динамическое профилирование
  - Подмена кода
  - Параллельная интерпретация байт-кода и выполнение бинарного кода
- За основу мы взяли LLVM (биткод и JIT уже есть)
- Добавили часть недостающих компонент
  - Динамическое профилирование
  - Параллельный JIT и выполнение кода
  - Не хватает горячей замены одной версии кода на другую (on-stack replacement)
  - Уже сейчас может быть быстрее обычной компиляции

Статический анализ. Выявление дефектов в программах

- Дефекты (ситуации в исходном коде) могут приводить к:
  - Уязвимостям защиты
  - Потере стабильности работы программы
- Статический анализ исходного кода:
  - Автоматический анализ многих путей исполнения одновременно
  - Обнаружение дефектов, проявляющихся только на редких путях исполнения, или на необычных входных данных (которые могут быть установлены злоумышленником в процессе атаки)
- Динамические проверки не работают: инструментированный код может работать на порядок медленнее исходного
- Трудности статического анализа:
  - Ложные срабатывания
    - Принципиальные ограничения алгоритмов анализа
    - Неполнота информации о программе (отсутствие исходного кода библиотек и т.д.)
  - Производительность алгоритмов анализа
    - Возможность анализа больших программ

Пример (ошибки форматных строк)

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buf[80], cmd[100];
    fgets (buf, sizeof(buf), 80);
    snprintf (cmd,sizeof(cmd),"ls -l %s",buf);
    system (cmd);
    return 0;
}
```

Например: **ls -l myfile ; rm -rf /**

Автоматическое обнаружение дефектов

- Поиск дефектов не обязательно исчерпывающий
- Нет ограничений на анализируемый исходный код
- Изменение исходного кода не требуется
- Поиск ситуаций в исходном коде, указывающих на наличие дефектов
- Существующие коммерческие системы:
  Coverity, Klocwork, CodeSonar

- Наша система:
  - Алгоритм основан на анализе алиасов, def-use анализе, анализе потока данных
  - Автоматическое построение аннотаций функций
  - Использование аннотаций при межпроцедурном распространенib атрибутов значений программы снизу вверх
  - Набор подсистем, реализующих поиск конкретных ситуаций в программе, указывающих на наличие дефектов
  - Подсистемы используют общий механизм распространения атрибутов, указывая интересующие их свойства значений программы и правила распространения атрибутов

Сравнение с Coverity Prevent

Тип предупреждения	Истинные срабатывания (наши)	Истинные срабатывания Prevent	Воспроизведено истинных срабатываний Prevent, %
Переполнение буфера	60%	10%	100%
Работа с динамически выделенной памятью	50%	70%	20%
Разыменование NULL	70%	60%	50%
Испорченный ввод	70%	70%	80%
Неинициализированные данные	60%	40%	50%
Несоотвестие типов возвращаемых значений	60%	90%	30%
Состояние гонки	90%	90%	80%
Передача по значению	100%	100%	100%
Другие (более 30)	50%	70%	30%

Что нужно знать для работы с нами

- Ориентироваться в любой *nix-системе
- Уметь пользоваться системой контроля версий svn/git, отладчиком gdb, системой сборки make, и gcc
- Обладать знаниями об устройстве компиляторов
  - Курсы кафедры по конструированию компиляторов, спецкурс по устройству оптимизирующих компиляторов
  - На спецсеминаре проходит знакомство с устройством оптимизирующей части компиляторов и современными оптимизациями
  - Выполняются курсовые и дипломные работы по улучшению существующих / написанию новых оптимизаций, анализу программ
  - Выполняются работы в рамках проектов, над которыми работает группа

Источники информации

- Альфред Ахо, Рави Сети, Джеффри Ульман "Компиляторы: Принципы, Технологии, Инструменты“. Он же Compilers: Principles, Techniques, and Tools (2nd Edition), by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- Engineering a Compiler, by Keith Cooper, Linda Torczon
- http://www.hipeac.net/node/746 – GCC Tutorial at HiPEAC Workshop
- http://www.gccsummit.org
- http://gcc.gnu.org/onlinedocs/gccint/
- http://gcc.gnu.org/wiki/
- http://llvm.org