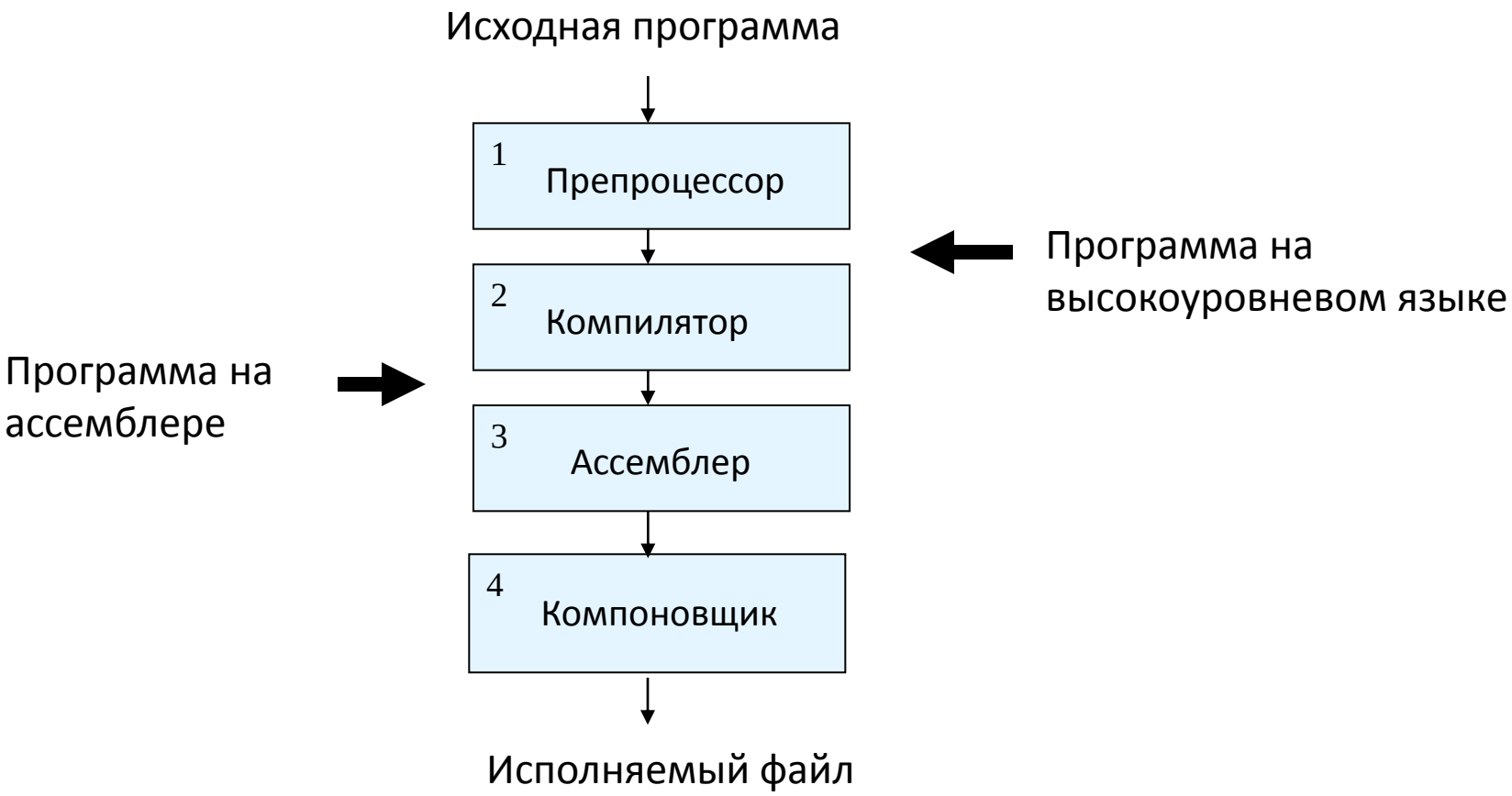
ИСП РАН
Институт системного программирования
Российской академии наук

Оптимизации в статических и динамических компиляторах и их применение на примере GCC, LLVM, JavaScript-движков и PostgreSQL

Дмитрий Мельник

dm@ispras.ru

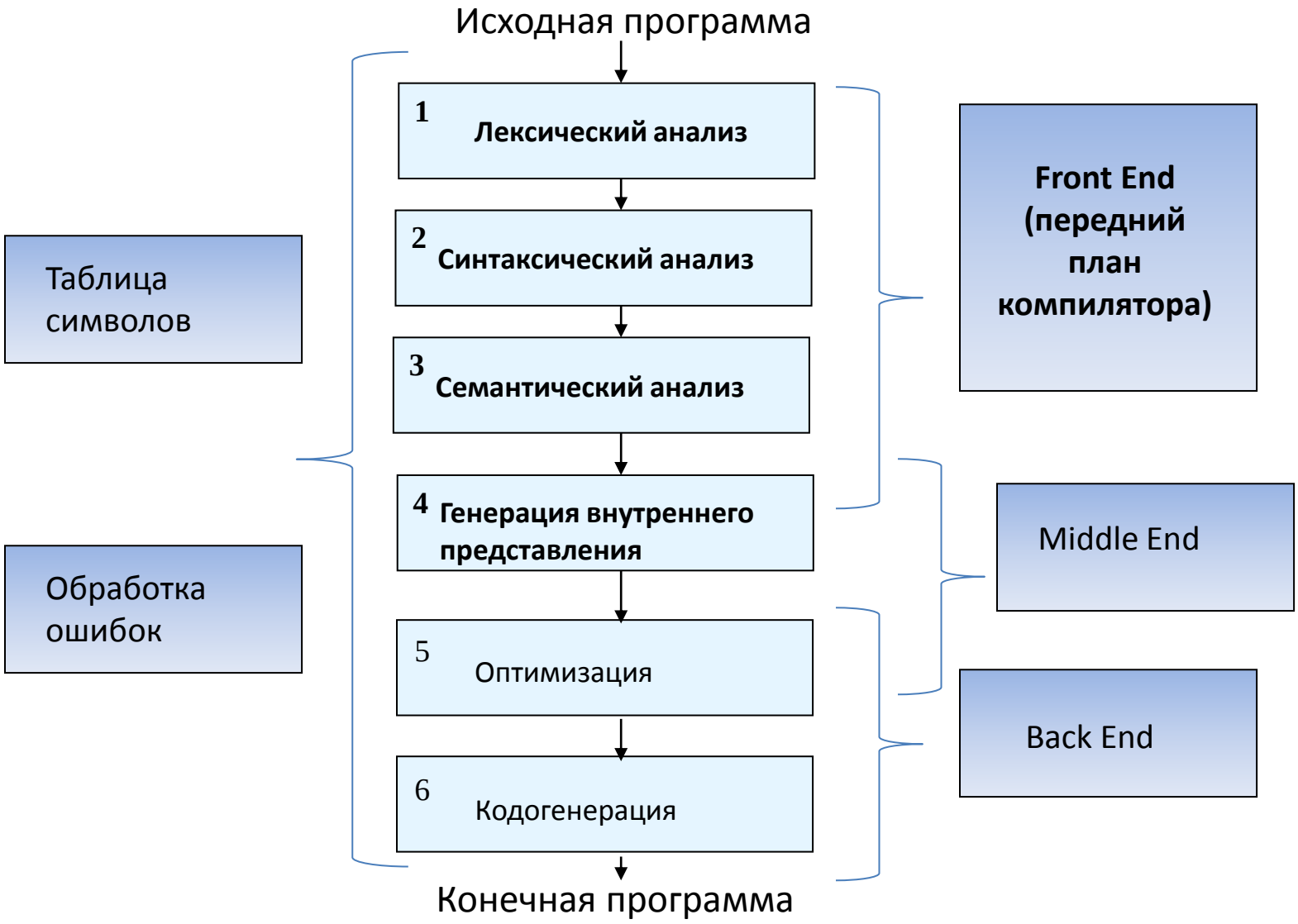
Цепочка компиляции



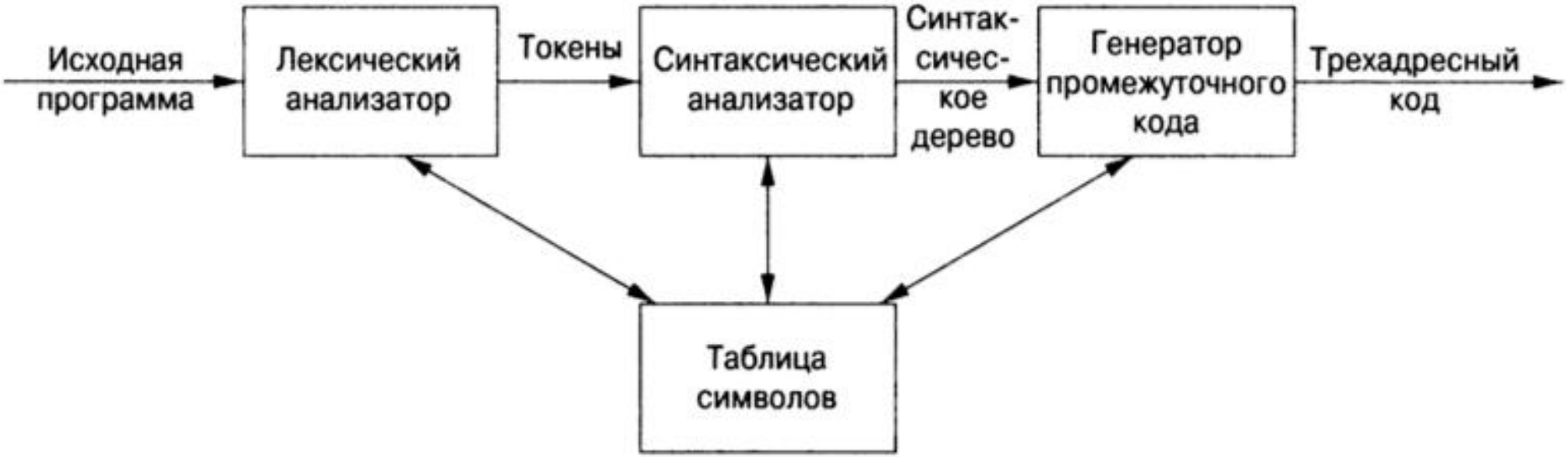
Препроцессирование

- 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))

Фазы компиляции



Структура front-end’a

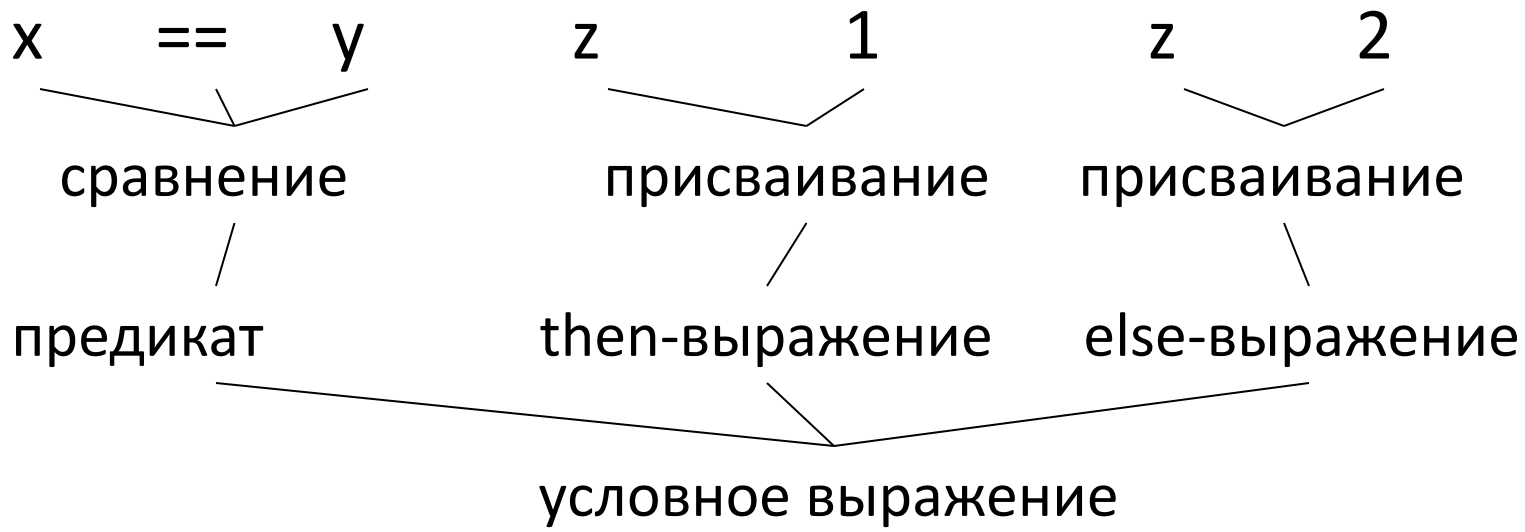


Лексический анализ

- 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 = 1e10; else z = 3.14;
  - if, x, ==, y, then, z, =, 1e10, ;, else, z, =, 3.14, ;

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

- 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
- Проверка соответствия типов

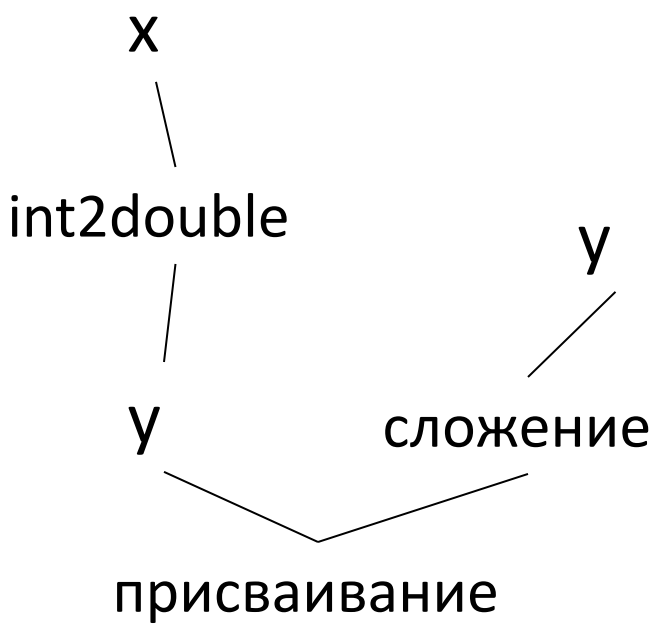
```
{
    char Jack[128];
    {
        int Jack = 4;
        printf("%d\n", Jack);
    }
}
```

Семантический анализ

- Результирующее дерево может содержать дополнительные операции, вставленные на этом этапе

```
int x, double y;

y =  x + y;
```



- Таблица имен заполняется в ходе всего анализа, начиная с лексического

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

Оптимизация программы

- Наиболее интересная и “наукоемкая” часть компилятора
- 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 (динамические)

Транслятор

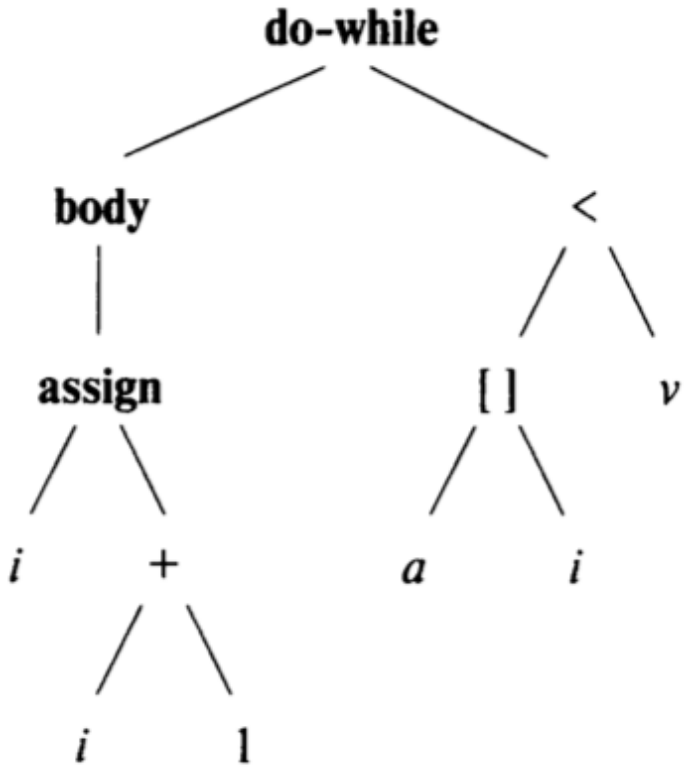
```
{
  int i; int j; float[100] a;
  float v; float x;
  while ( true ) {
    do i = i+1; while ( a[i] < v );
    do j = j-1; while ( a[j] > v );
    if ( i >= j ) break;
    x = a[i];
    a[i] = a[j];
    a[j] = x;
  }
}
```

Исходная программа

```
1: i = i + 1
2: tl = a [ i ]
3: if tl < v goto 1
4: j = j - 1
5: t2 = a [ j ]
6: if t2 > v goto 4
7: ifFalse i >= j goto 9
8: goto 14
9: x = a [ i ]
10: t3 = a [ j ]
11: a [ i ] = t3
12: a [ j ] = x
13: goto 1
14:
```

Результат трансляции: промежуточное представление (трехадресный код)

Трансляция во внутреннее представление



Абстрактное Синтаксическое Дерево (AST)

Исходная программа

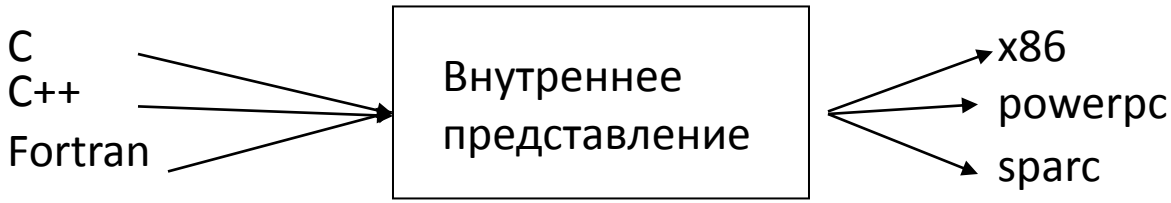
```
do {
   i = i + 1 ;
} while (a[i] < v);"
```

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

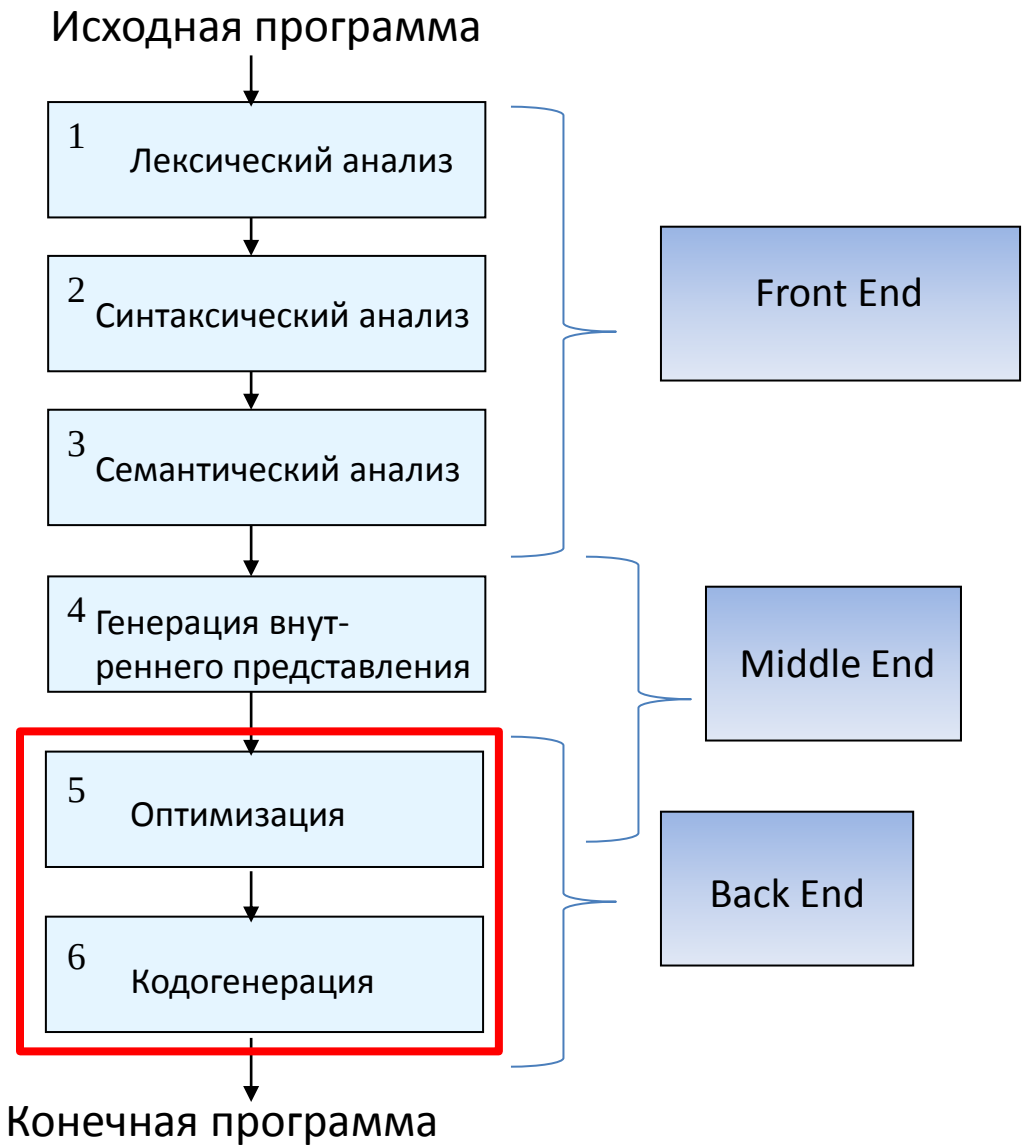
Результат трансляции: промежуточное представление (трехадресный код)

Зачем нужно внутреннее представление

- Удобство выполнения анализа и синтеза
- Возможность построить компилятор с нескольких языков на несколько архитектур (gcc)
- Для разных этапов – разные представления



Фазы компиляции



- Машинно-независимые оптимизации
- Машинно-зависимые оптимизации

Middle End или Back End?

- Вообще говоря, разделение на middle end/back end и машинно-зависимые/независимые оптимизации условно. В GCC удобнее проводить подобное разделение по тому, на каком внутреннем представлении работают оптимизации

- После распределения регистров внутреннее представление становится машинно-зависимым: появляется 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;
```

Машинно-независимые оптимизации

- Внутреннее представление – обычно трехадресное
  - Временные переменные (псевдорегистры) и память
  - Сохраняется знание о высокоуровневых типах

```
a[i] = a[j]*3 + a[k];
```

```
t1 = a[j]; t2 = t1 * 3;
t3 = a[k]; t4 = t2 + t3;
a[i] = t4;
```

- Примеры машинно-независимых оптимизаций:
  - Удаление мертвого кода (кода, который не влияет на результат)
  - Скалярные оптимизации (упрощение выражений)
  - Оптимизации циклов (над массивами)
  - Встраивание функций (inlining)
  - Продвижение констант (constant propagation)
  - Трансформация циклов (loop transformation: loop unrolling, peeling, distribution, fusion)

Встраивание функций (inlining)

```
class Bar {
  int size;
  Bar(int _size) {
    size = _size;
  }
  int getSize() {
    return size;
  }
}
Bar a(10), b(20);
int x = a.getSize() +
    b.getSize();
```

```
int sum = a.size + b.size;
```

```
int sum = 30;
```

- Встраивание особенно эффективно для C++ кода, где часто используются небольшие функции для доступа к полям классов

Встраивание функций (inlining)

```
int foo(int a, int b, int c, int d,
        int e)
{
  return a + b + c + d + e;
}

int main() {
  return foo(1, 2, 3, 4, 5);
}
```

```
int main() {
  return 1 + 2 + 3 + 4 + 5;
}
```

- Увеличится ли размер кода после выполнения оптимизации? Станет ли быстрее? Что нужно учитывать, чтобы вычислить это?
- По умолчанию, в GCC встраивание выполняется, только если в результате размер кода уменьшится.

Встраивание функций (inlining)

```
void bar();

int foo(int a, int b, int c, int d,
        int e)
{
  int arr[100];
  …
  bar();
  return a + b + c + d + e;
}

int main() {
  return foo(1, 2, 3, 4, 5);
}
```

Пролог:

- Сохранение регистров, которые «портит» функция
- Сохранение и установка stack pointer

Эпилог:

- Восстановление регистров и stack pointer
- Возврат из функции

Вызов функции:

- Подготовка аргументов

```
foo:
        .cfi_startproc
Пролог  movq    %rbx, -32(%rsp)
        movq    %rbp, -24(%rsp)
        movq    %r12, -16(%rsp)
        movq    %r13, -8(%rsp)
        subq    $440, %rsp
        ...
Эпилог  movq    408(%rsp), %rbx
        movq    416(%rsp), %rbp
        movq    424(%rsp), %r12
        movq    432(%rsp), %r13
        addq    $440, %rsp
        ret

main:
        ...
Вызов   movl    $5, %r8d
функции movl    $4, %ecx
        movl    $3, %edx
        movl    $2, %esi
        movl    $1, %edi
        call    foo
```

x86_64

Встраивание функций (inlining)

```
void bar();

int foo(int a, int b, int c, int d,
        int e)
{
  int arr[100];
  …
  bar();
  return a + b + c + d + e;
}

int main() {
  return foo(1, 2, 3, 4, 5);
}
```

Пролог:

- Сохранение регистров, которые «портит» функция, а также адреса возврата (в **LR**)
- Сохранение и установка stack pointer

Эпилог:

- Восстановление регистров и stack pointer
- Возврат из функции (lr записывается в pc)

Вызов функции:

- Подготовка 1-х четырех аргументов в регистры, 5-го в стек

```
Пролог:
foo:
        stmfd   sp!, {r4, r5, r6, lr}
        sub     sp, sp, #400
        …
        ldr     r4, [sp, #416]
        …
Эпилог:
        add     sp, sp, #400
        ldmfd   sp!, {r4, r5, r6, pc}
        bx      lr

main:
        …
Вызов функции:
        mov     r3, #5
        str     r3, [sp]
        mov     r0, #1
        mov     r1, #2
        mov     r2, #3
        mov     r3, #4
        bl      foo
```

ARMv7

Насколько машинно-независимые оптимизации все-таки зависят от архитектуры?

- Существует API, через который машинно-независимые оптимизации могут запрашивать необходимые параметры у целевой платформы, например:
  - стоимость инструкций (выраженную в размере кода или в количестве необходимых пересылок);
  - Поддержка аппаратных особенностей: условное выполнение, предзагрузка, величина выравнивания
  - Доступность функциональных устройств процессора (моделирование конвейера)

Машинно-зависимые оптимизации

- Внутреннее представление становится машинно-зависимым
  - Адресная арифметика, указатель на стек, регистры
- 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;
- Оптимизации, специфичные для данной архитектуры
  - Peephole (замена соседних команд на более эффективный вариант)
  - Оптимизация кодировки (Thumb-2)
  - Выбор шаблонов инструкций (IA-64)

Уровни параллелизма в процессоре

- **Конвейер**
  - Позволяет параллельно исполнять инструкции, находящиеся на разных стадиях исполнения
- **Параллелизм функциональных устройств**
  - Несколько конвейеров / функциональных устройств

Конвейер ARM

- Cortex-A8, A7
  - Конвейер из 13 стадий
  - 2 АЛУ устройства, 1 Load/Store, 1 Multiply
  - Может выполнять до 2-х команд за такт
  - Из них должно быть не более одной Load/Store и Multiply, причем умножение должно идти первым
- Cortex-A9, A15, A57: используется динамическое переупорядочение команд

Примеры расписания выполнения команд на конвейере

- **ADD r1, r2, r3**
  - r2, r3: требуются перед E2
  - r1: готов после E2
- **MOV r1, r2 asl #const**
  - r2: требуется перед E1
  - r1: готов после E1
- Такие две команды могут начать выполняться на одном такте:

	E1	E2
`mov r1, r2`	R2	
`add r3, r1, r2`	R2	R1, R2 R3

← *Стадии конвейера*

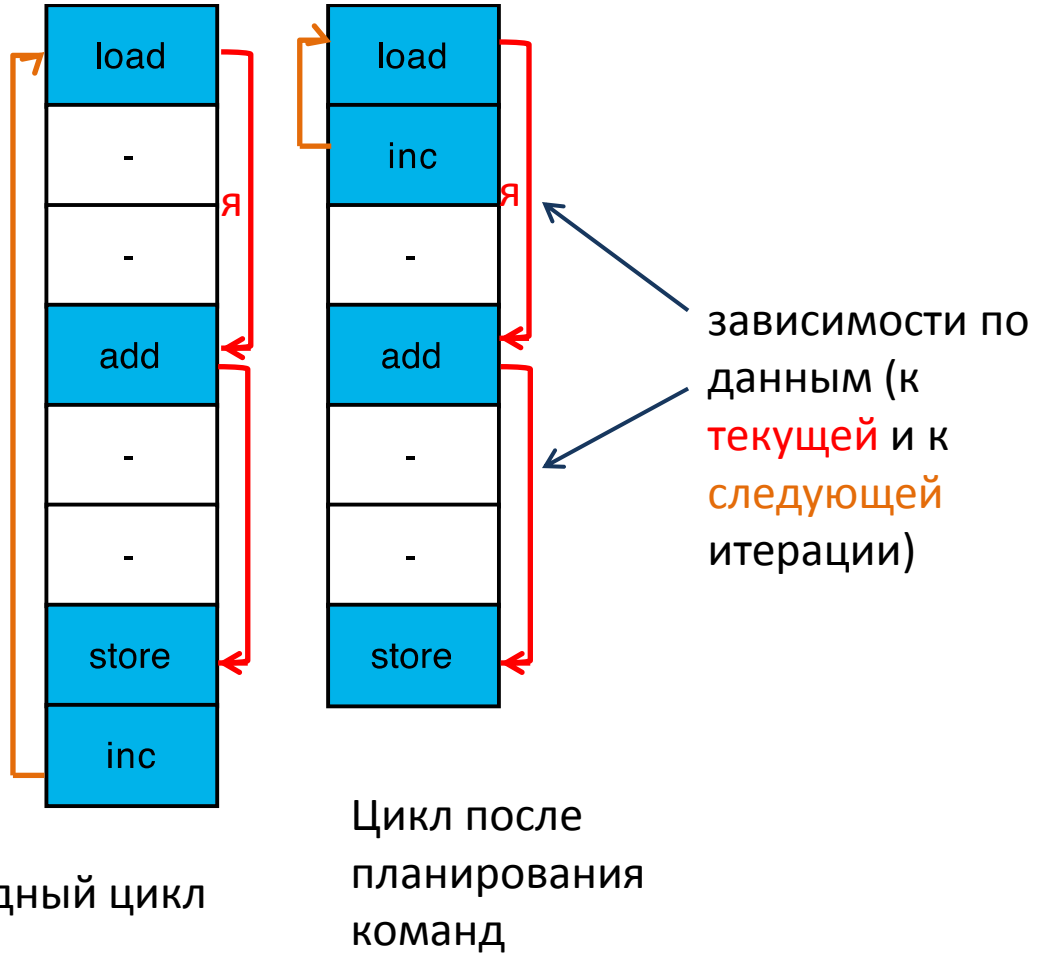
- При выдаче этих команд в обратном порядке появится задержка в 2 такта:

	E1	E2	E3
`add r3, r1, r2`	--	R1, R2 R3	
`mov r4, r3`	Ожидание готовности R2		R3 R4

Обозначения:
Rn – Регистр требуется в начале стадии выполнения Ei
Rk – Записывается в конце стадии Ei

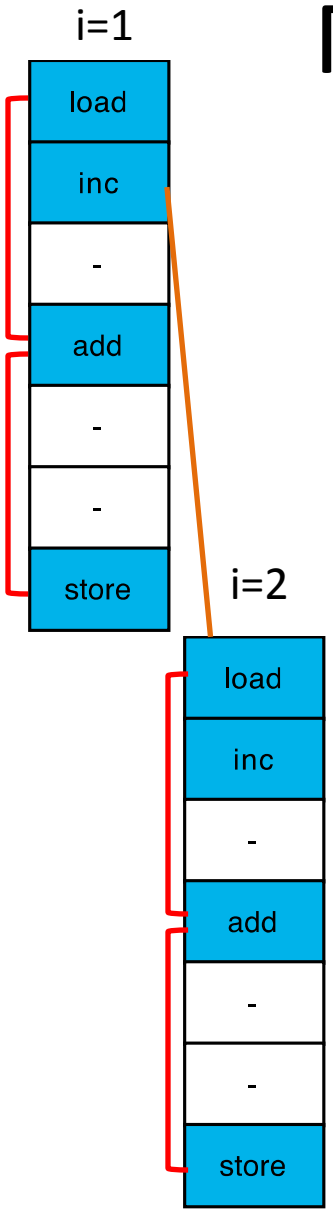
Планирование команд

```
loop:
  arr[i] = arr[i] + 1
  i++
  goto loop
```

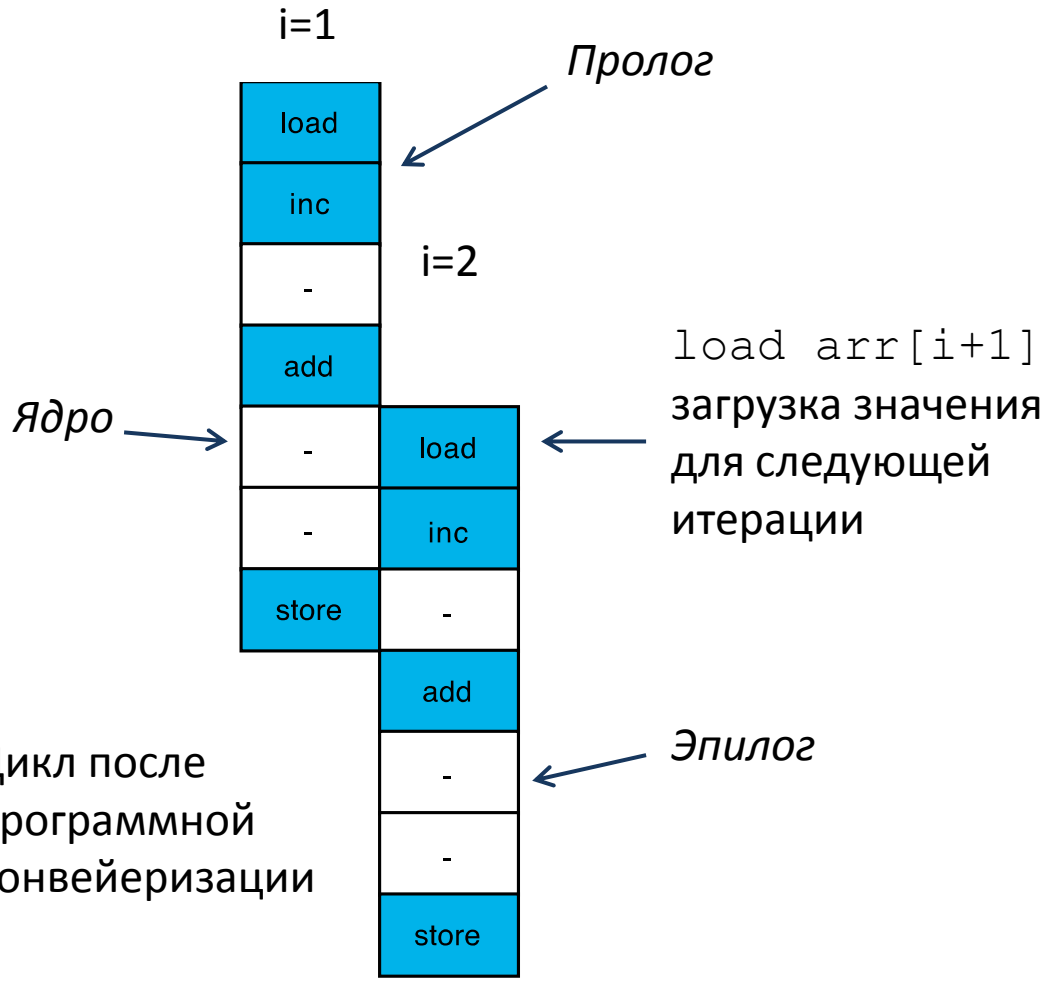


Задача: требуется минимизировать длину расписания так, чтобы зависимости по данным между командами были удовлетворены

Программная конвейеризация циклов



Цикл после планирования команд



`load arr[i+1]` загрузка значения для следующей итерации

Цикл после программной конвейеризации

Селективный планировщик

-O2

```
.L11:
        ldr     ip, [r3, r7]
        ldr     r4, [r3, r6]      } 2 cycles
        add     r3, r3, #4
        cmp     r3, r5
        mla     r2, r4, ip, r2    } 5 cycles
        mul     r1, r2, r1
        bne     .L11
```

До:
12 тактов

Source

```
int foo(int N) {
  int i;
  int a=0, b=1;
  for (i=0; i<N; i++)
  {
    a += x[i] * y[i];
    b *= a;
  }
  return a+b;
}
```

Selective Scheduling

```
Prologue:
        ldr     r4, [r3, r7]
        mov     ip, #1
        ldr     r5, [r3, r8]
        mov     r2, r3
.L15:
Kernel:
        mla     r2, r5, r4, r2
        cmp     r1, r6
        mov     r3, r1
        ldrne   r4, [r3, r7]
        addne   r1, r3, #4
        ldrne   r5, [r3, r8]
        mul     ip, r2, ip
        bne     .L15
```

После:
8 тактов

Ускорение
8%

Селективный планировщик выполняет конвейеризацию циклов

- «Скрывает» задержки у долгих команд, перенося их на предыдущую итерацию цикла
- Помогает использовать параллелизм на уровне команд
- Среднее ускорение 1-3%

Улучшения кодогенерации для ARM Advanced SIMD (NEON)

- **Более полная поддержка команд NEON**

  Например, поддержка комбинирования $|a - b|$ в одну команду NEON:

  `vsub`
  `vabs` → `vabd`

  Производительность: **+2.5%** (x264)
  Уменьшение размера: **0.1%** (x264)

- **Улучшение распределения регистров для векторов констант**

  Запрет на размещение векторных констант в обычных ARM регистрах

  Производительность: **+3%** (evas)

- **Поддержка преобразования из `float` в `int` в векторизации**

  Циклы, в которых используется преобразование типов, теперь могут быть векторизованы с использованием команды `vcvt`

  Производительность: **+9%** (libmp3lame)

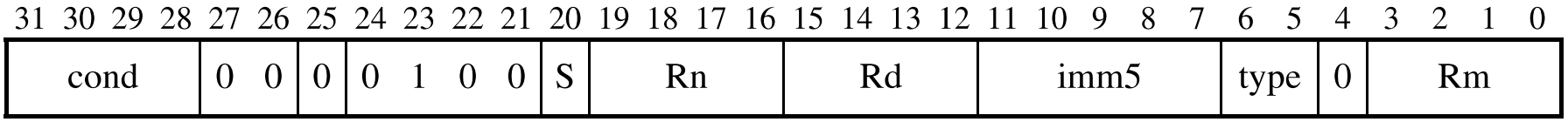
Кодировка команд

ADDGE r0, r1, r2 asr #31

Действие:

```
if (flags_match (GE))
    r0 = r1 + r2 >> 31;
```

ADD{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}



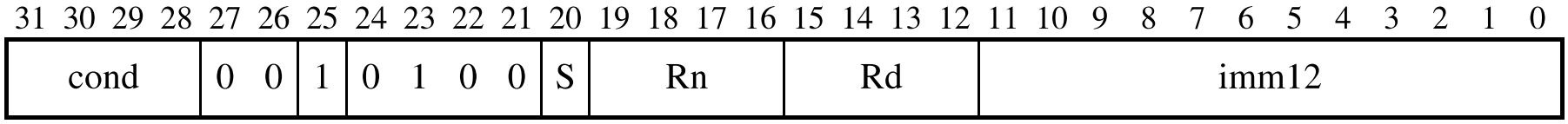
GE ALU ADD 0 1 0 31 ASR 2
(w/reg)

BICS r0, r1, #FF000000

Действие:

```
r0 = r1 & ~0xff000000;
set_flags();
```

ADD{S}<c> <Rd>,<Rn>,#<const>



Представление констант

Второй аргумент ALU-команд может быть также константой, которая кодируется 12 битами (8 бит значение, 4 бита величина сдвига):

*CONST_32 = CONST_8 << (2 * N), 0 <= N < 16*

Примеры правильных и неправильных констант:

```
and r1, r1, #255
and r1, r1, #510
and r1, r1, #0xff00ff00
and r1, r1, #0xff000000
```

Представление констант

Второй аргумент ALU-команд может быть также константой, которая кодируется 12 битами (8 бит значение, 4 бита величина сдвига):

*CONST_32 = CONST_8 << (2 * N), 0 <= N < 16*

Примеры правильных и неправильных констант:

```
and r1, r1, #255
and r1, r1, #510 // 510 = 255 << 1 – нечетный сдвиг
and r1, r1, #0xff00ff00 // значение «шире» 8 бит
and r1, r1, #0xff000000
```

Улучшение оптимизации GCSE и комбинирования команд в GCC

Исходный пример

```
if (x)
  a = b + c << 2;
else
  d = e + c << 2;
```

Было в GCC (неверно)

```
  mov r1, r2, asl #2
  cmp r3, #0
  bne L1
  add r4, r5, r1
  b .L2
.L1:
  add r6, r7, r1
```

После исправления

```
  cmp r3, #0
  bne L1
  add r4, r5, r2, asl #2
  b .L2
.L1:
  add r6, r7, r2, asl #2
```

Проблема:

- Оптимизация GCSE (Global Common Subexpression Elimination) «не знает» о возможности ARM barrel shifter
- Комбинирование команд (которое о нем «знает») работает только в предлах базового блока

Решение:

- Исправить GCSE
- Доработать оптимизацию комбинирования команд

Результаты:

- Сокращение размера кода: **3.6 Kb** на SPEC 2K INT (**0.1%**)
- Используется меньше регистров

Режимы процессора ARM

	ARM	Thumb-1	Thumb-2
Размер команды	32 бит	16 бит	16/32 бит
Кол-во команд	~60	30	~60
Условное выполнение	Почти все команды	Нет	С помощью IT-блоков
Сдвиг аргументов	Во всех командах ALU	Отдельные команды	В 32-битных командах
Доступные регистры	15 общего назначения + pc	8 общего назначения + 7 специальных + pc	15 общего назначения + pc
Примеры команд	`add r1, r2, r3 asl #2` **(A)**	`add r1, r2` **(B)**	Обе формы допустимы; Размер **(A)** – 16 бит, **(B)** – 32 бит

Улучшения обработки условных команд Thumb-2

Проблемы:

- Команда внутреннего представления RTL `if-then-else` раскрывается напрямую в ассемеблер слишком поздно
- Слишком ранняя оптимизация коротких Thumb-2 команд мешает последующему преобразованию в условную форму
- IT блоки могут быть разделены в планировщике

Решения:

- Раньше раскрывать `if-then-else` в RTL
- Изменить порядок оптимизаций в GCC
- Отдавать приоритет командам с тем же предикатом в планировщике

RTL псевдо-код

```
a = (x == 0) ? 1 : 2;
b = (x == 0) ? 3 : 4;
```

До

```
cmp r1, #0
ite eq
moveq r2, #1
movne r2, #2
ite eq
moveq r3, #3
movne r3, #4
```

После

```
cmp r1, #0
itete eq
moveq r2, #1
movne r2, #2
moveq r3, #3
movne r3, #4
```

Улучшения обработки условных команд Thumb-2

Другие улучшения:

- **Преобразовывать максимум 4 команды, чтобы не увеличивать код**
- Не преобразовывать команды в условную форму, если вероятность одной дуги значительно выше другой (напр. 90% к 10%)

Исходный пример

```
  cmp r1, #0
  bne .L2
  mov r1, #1
  mov r2, #2
  mov r3, #3
  mov r4, #4
  mov r5, #5
.L2:
  ...
```

Преобразованный в условную форму

```
  cmp r1, #0
  itttt eq
  moveq r1, #1
  moveq r2, #2
  moveq r3, #3
  moveq r4, #4
  it eq
  moveq r5, #5
  ...
```

Условный переход удаляется, один IT блок - ОК

Преобразование более 5 команд «стоит» лишней IT-команды

Улучшения обработки условных команд Thumb-2

Другие улучшения:

- Преобразовывать максимум 4 команды, чтобы не увеличивать код
- **Не преобразовывать команды в условную форму, если вероятность одной дуги значительно выше другой (напр. 90% к 10%)**

Если вероятность перехода высокая, то лучше полагаться на branch prediction

Исходный пример

```
  cmp r1, #0
  bne .L2
  mov r1, #1
  mov r2, #2
  mov r3, #3
  mov r4, #4
  mov r5, #5
.L2:
  ...
```

Преобразованный в условную форму

```
cmp r1, #0
itttt eq
moveq r1, #1
moveq r2, #2
moveq r3, #3
moveq r4, #4
it eq
moveq r5, #5
...
```

Иначе конвейер может быть заполнен кодом, который никогда не выполняется

Использование сокращенной кодировки Thumb-2

32 bit:

```
ADD R9, R10, R11
```

16 bit:

```
ADDS R1, R2, R3
```

arm_reorg:
Если регистр флагов не используется, а номера регистров < 8, то добавить суффикс “S”. тогда команда станет 16-битной

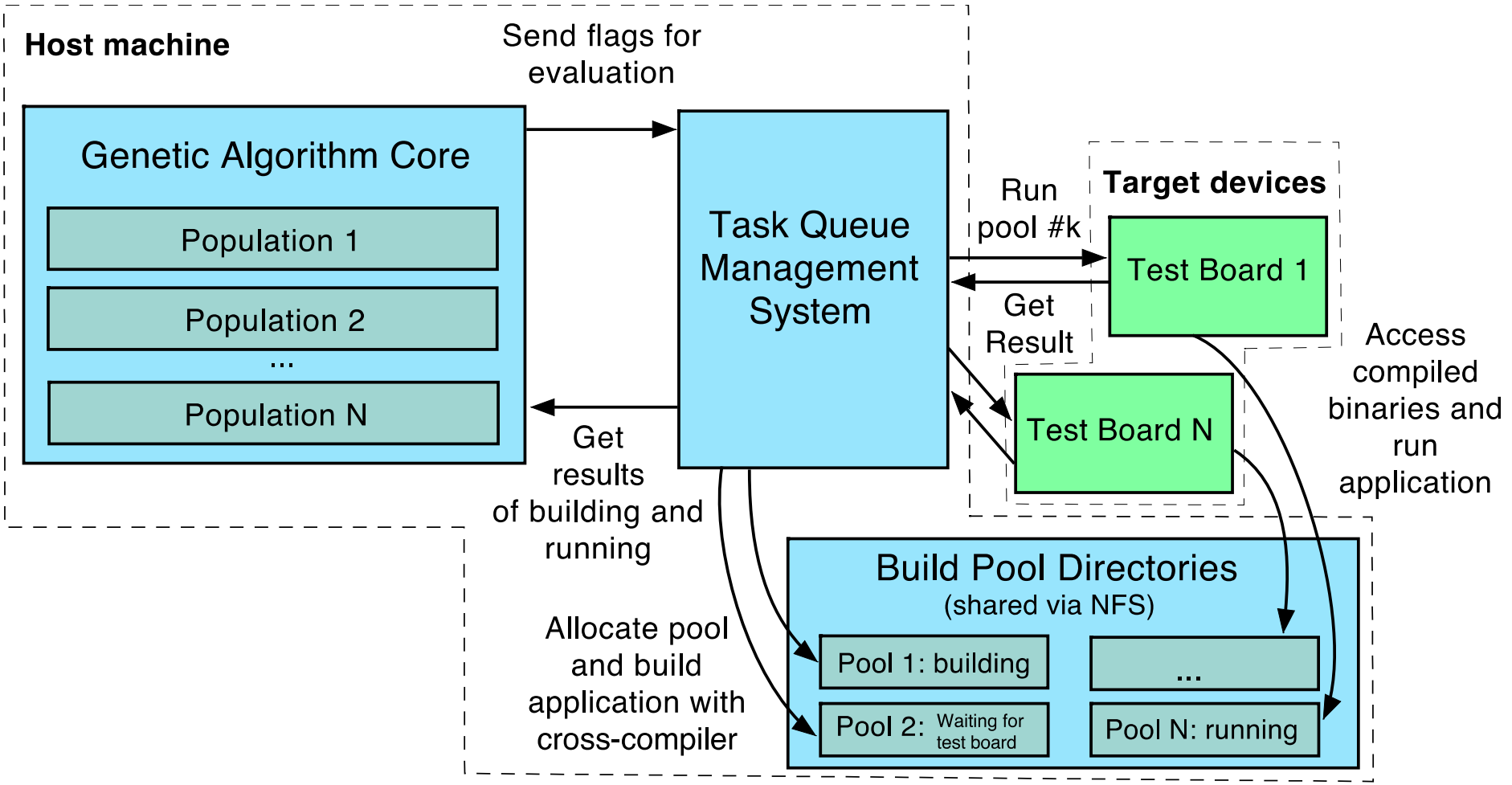
новая оптимизация:
Если кодировка команды допускает 16-битную версию при условии, что номера регистров < 8, то попытаться распределить команде регистры с маленькими номерами

TACT

TACT: Tool for Automatic Compiler Tuning

- Motivation
  - GCC is multiplatform compiler, has 150+ optimizations which are not always tuned well for the specific platform
  - Goal: improve the performance of GCC for ARM
  - Traditional approach: profile application, then find out what needs to be improved in the compiler
  - It's difficult, and takes much time and expertise
- Performance analysis involving TACT:
  - Use Genetic Algorithm to find best-performing compile options
    - Also find default -O2 optimizations that make code worse
  - Find which of the options impact performance the most
  - Compare the assembly
  - Fix relevant optimization in GCC to make what we've found the compiler's default behavior

How TACT Works



Tuning Results (raw)

-O2 -mcpu=cortex-a9 -mtune=cortex-a9 -mfpu=neon -mfloat-abi=softfp -fno-aggressive-loop-optimizations -fno-align-functions -fno-align-jumps -fno-align-labels -falign-loops -fno-argument-alias -fno-argument-noalias -fargument-noalias-anything -fno-argument-noalias-global -fno-associative-math -fno-asynchronous-unwind-tables -fno-auto-inc-dec -fbranch-count-reg -fbranch-probabilities -fno-branch-target-load-optimize -fno-branch-target-load-optimize2 -fno-btr-bb-exclusive -fcaller-saves -fcheck-data-deps -fno-combine-stack-adjustments -fno-common -fcompare-elim -fconserve-stack -fno-cprop-registers -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fno-data-sections -fdce -fdefer-pop -fno-delayed-branch -fdelete-dead-exceptions -fdelete-null-pointer-checks -fdevirtualize -fno-dse -fearly-inlining -fno-expensive-optimizations -ffast-math -ffinite-math-only -ffloat-store -fno-forward-propagate -ffunction-cse -fno-function-sections -fgcse -fgcse-after-reload -fgcse-las -fgcse-lm -fgcse-sm -fguess-branch-probability -fhoist-adjacent-loads -fif-conversion -fno-if-conversion2 -fno-indirect-inlining -finline -finline-functions -finline-functions-called-once -finline-small-functions -fipa-cp -fipa-cp-clone -fno-ipa-matrix-reorg -fipa-pta -fipa-pure-const -fno-ipa-reference fipa-sra -fipa-struct-reorg -fivopts -fno-jump-tables -fkeep-inline-functions -fkeep-static-consts -floop-nest-optimize -fmerge-all-constants -fmerge-constants -fmodulo-sched -fno-modulo-sched-allow-regmoves -fmove-loop-invariants -fno-optimize-register-move -foptimize-sibling-calls -foptimize-strlen -fno-pack-struct -fno-partial-inlining -fno-peel-loops -fpeephole -fpeephole2 -fpredictive-commoning -fno-prefetch-loop-arrays -fno-reciprocal-math -free -fno-reg-struct-return -fno-regmove -frename-registers -fno-reorder-functions -fno-rerun-cse-after-loop -freschedule-modulo-scheduled-loops -frounding-math -fno-sched-critical-path-heuristic -fno-sched-dep-count-heuristic -fno-sched-group-heuristic -fno-sched-interblock -fno-sched-last-insn-heuristic -fsched-pressure -fno-sched-rank-heuristic -fno-sched-spec -fsched-spec-insn-heuristic -fno-sched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns -fno-sched-stalled-insns-dep -fsched2-use-superblocks -fsched2-use-traces -fno-schedule-insns -fschedule-insns2 -fshrink-wrap -fno-section-anchors -fno-selective-scheduling -fselective-scheduling2 -fsel-sched-pipelining -fno-sel-sched-pipelining-outer-loops -fno-sel-sched-reschedule-pipelined -fsee -fno-signed-zeros -fno-single-precision-constant -fsplit-ivs-in-unroller -fsplit-wide-types -fno-stack-check -fno-thread-jumps -ftoplevel-reorder -fno-tracer -fno-trapping-math -fno-tree-bit-ccp -fno-tree-builtin-call-dce -fno-tree-ccp -ftree-ch -ftree-coalesce-vars -ftree-copy-prop -ftree-copyrename -ftree-cselim -ftree-dce -fno-tree-dominator-opts -fno-tree-dse -ftree-forwprop -ftree-fre -ftree-loop-distribution -ftree-loop-distribute-patterns -fno-tree-loop-if-convert -ftree-loop-if-convert-stores -fno-tree-loop-im -ftree-loop-ivcanon -fno-tree-loop-optimize -ftree-lrs -ftree-partial-pre -fno-tree-phiprop -fno-tree-pre -ftree-pta -ftree-reassoc -ftree-scev-cprop -ftree-sink -fno-tree-slp-vectorize -ftree-slsr -fno-tree-sra -fno-tree-switch-conversion -fno-tree-tail-merge -fno-tree-ter -fno-tree-vect-loop-version -fno-tree-vectorize -fno-tree-vrp -fno-unroll-all-loops -fno-unroll-loops -fno-unsafe-loop-optimizations -fno-unsafe-math-optimizations -fno-unswitch-loops -fno-unwind-tables -fno-variable-expansion-in-unroller -fvect-cost-model -fno-vpt -fno-web -fno-ira-loop-pressure -fira-algorithm=CB -fira-region=all -fira-share-save-slots -fno-ira-share-spill-slots --param ira-max-loops-num=1000 --param inline-unit-growth=50 --param large-function-growth=100 --param large-function-insns=4800 --param large-stack-frame=264 --param large-stack-frame-growth=900 --param large-unit-insns=3000 --param max-delay-slot-live-search=669 --param max-inline-insns-auto=220 --param max-inline-insns-recursive=950 --param max-inline-insns-single=350 --param max-inline-recursive-depth=19 --param max-inline-recursive-depth-auto=17 --param max-variable-expansions-in-unroller=3 --param max-unrolled-insns=330 --param max-average-unrolled-insns=140 --param max-unroll-times=10 --param max-peeled-insns=360 --param max-peel-times=8 --param max-completely-peeled-insns=700 --param max-completely-peel-times=16 --param max-once-peeled-insns=680 --param min-inline-recursive-probability=7 --param simultaneous-prefetches=8 --param l1-cache-line-size=0 --param prefetch-latency=800 --param l1-cache-size=24 --param l2-cache-size=160 --param min-insn-to-prefetch-ratio=10 --param prefetch-min-insn-to-mem-ratio=0

Filtering the Results

- If compiling with and without a compiler option produces an identical binary, we throw it out
- Typically, only **15-30%** of the original options left in the resulting set
- Example of the reduced set of options (SPEC2K's `255.vortex` benchmark, 41 of 200 left):

-O2 -mcpu=cortex-a9 -mtune=cortex-a9 -mfpu=neon -mfloat-abi=softfp -fno-aggressive-loop-optimizations -fno-auto-inc-dec -fbranch-probabilities -fno-common -fconserve-stack -fno-cprop-registers -fno-dse -fno-expensive-optimizations -ffinite-math-only -ffloat-store -fno-forward-propagate -fgcse-after-reload -fgcse-las -fgcse-sm -fno-if-conversion2 -finline-functions -fipa-pta -fno-jump-tables -fkeep-inline-functions -fmodulo-sched -fno-partial-inlining -frename-registers -fno-reorder-functions -fno-rerun-cse-after-loop -fno-sched-interblock -fno-schedule-insns -fno-section-anchors -fselective-scheduling2 -fno-thread-jumps -fno-tree-ccp -fno-tree-dominator-opts -fno-tree-dse -fno-tree-loop-optimize -fno-tree-pre -fno-tree-sra -fno-tree-switch-conversion -fno-tree-ter -fno-tree-vrp -fira-algorithm=CB -fira-region=all -fno-ira-share-spill-slots --param ira-max-loops-num=1000 --param inline-unit-growth=50 --param large-stack-frame=264 --param large-stack-frame-growth=900 --param large-unit-insns=3000 --param max-inline-insns-auto=220 --param l1-cache-line-size=0

Example: Tuning Results for `255.vortex` from SPEC2K INT

Score	%Prev	%Base	Flags diff
10.720	0.00%	0.00%	-O2 -mcpu=cortex-a9 -mtune=cortex-a9 -mfpu=neon -mfloat-abi=softfp
10.340	3.54%	3.54%	-finline-functions
9.990	3.38%	6.81%	-fno-if-conversion2
10.050	-0.60%	6.25%	-fno-sched-interblock
9.870	1.79%	7.93%	-fselective-scheduling2
9.910	-0.41%	7.56%	-fno-tree-pre
9.730	1.82%	9.24%	--param max-inline-insns-auto=220
9.620	1.13%	10.26%	-fno-tree-dominator-opts
9.530	0.94%	11.10%	-fno-schedule-insns
9.490	0.42%	11.47%	-frename-registers
9.550	-0.63%	10.91%	-fno-forward-propagate
9.360	1.99%	12.69%	-fbranch-probabilities
9.330	0.32%	12.97%	-fno-rerun-cse-after-loop
9.340	-0.11%	12.87%	-fgcse-las -fno-tree-loop-optimize
9.280	0.64%	13.43%	-fno-tree-vrp
9.340	-0.65%	12.87%	-fno-section-anchors
9.290	0.54%	13.34%	-fno-tree-ter
9.260	0.32%	**13.62%**	--param large-unit-insns=3000

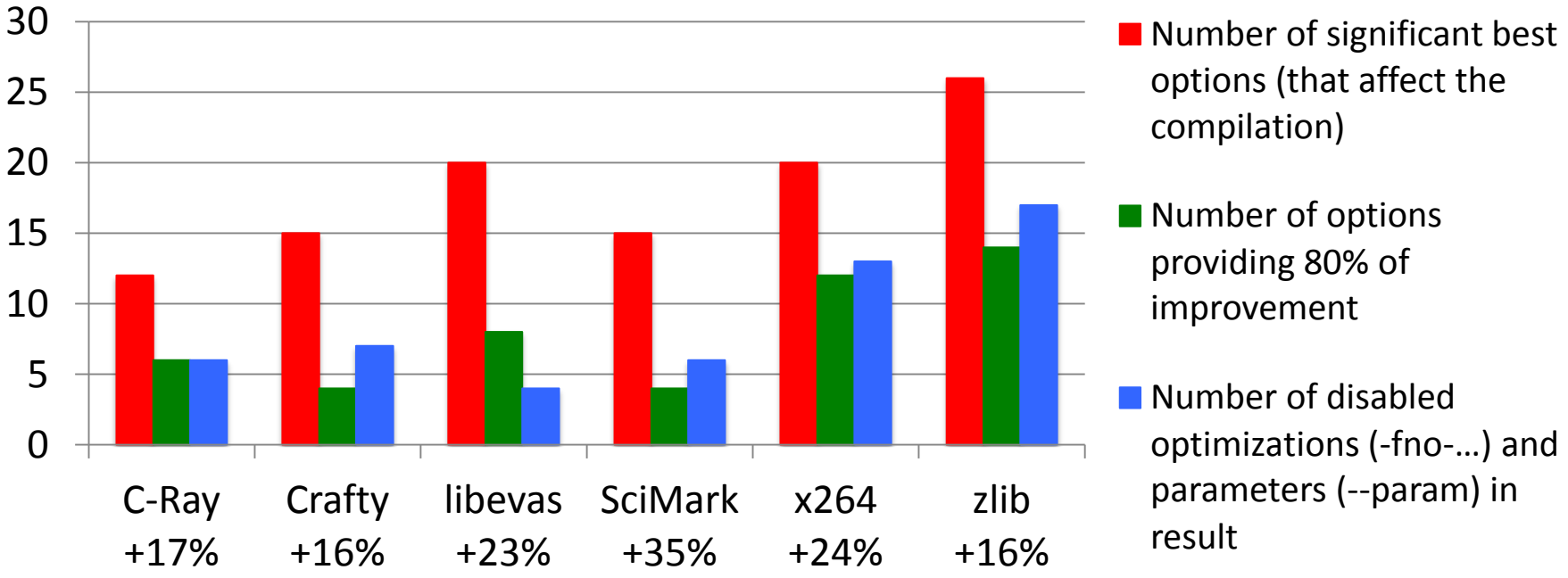
...

RED: default -O2 optimizations that don't work properly (or parameters that may have incorrect value) with significant impact

GREEN: candidate optimizations to enable by default for -O2 on this target

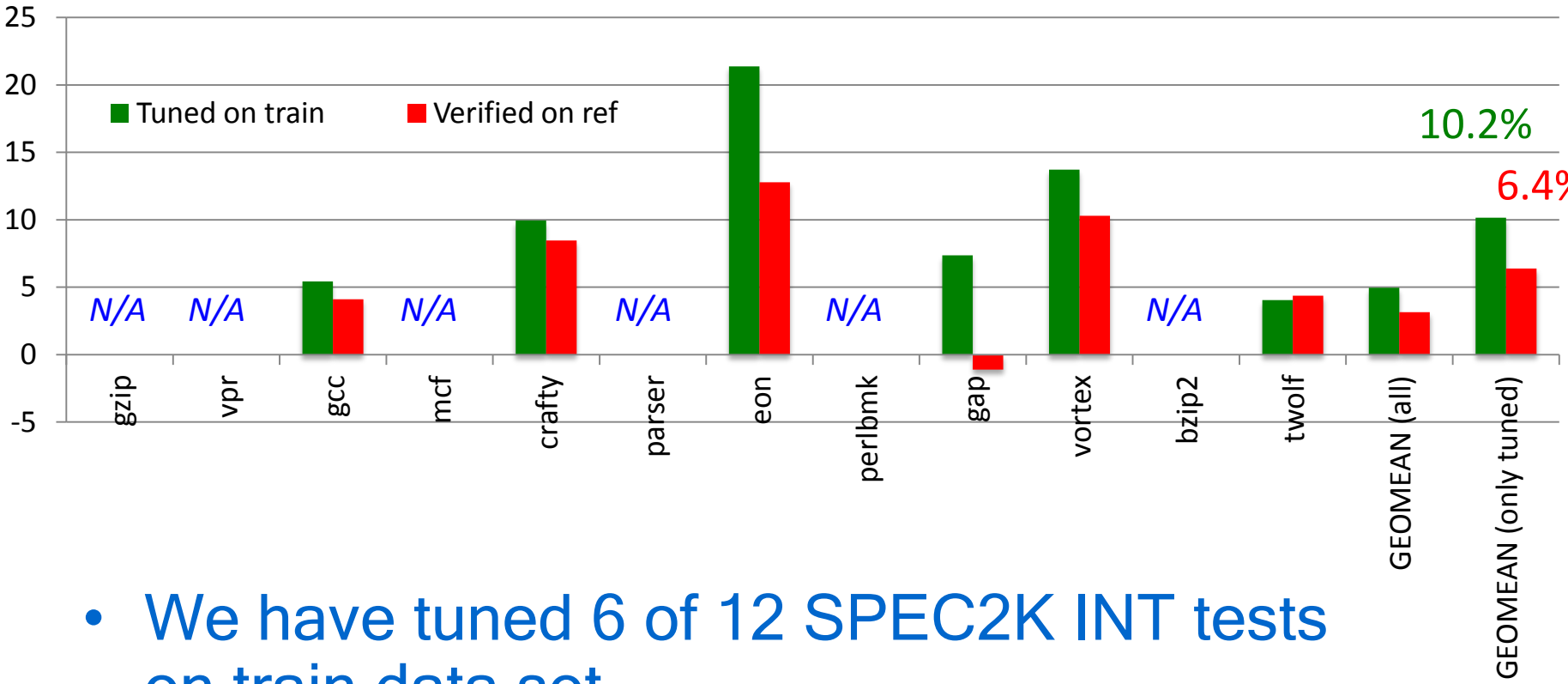
Tuning of Selected Open-Source Apps

- 200 original GCC options, 30 generations



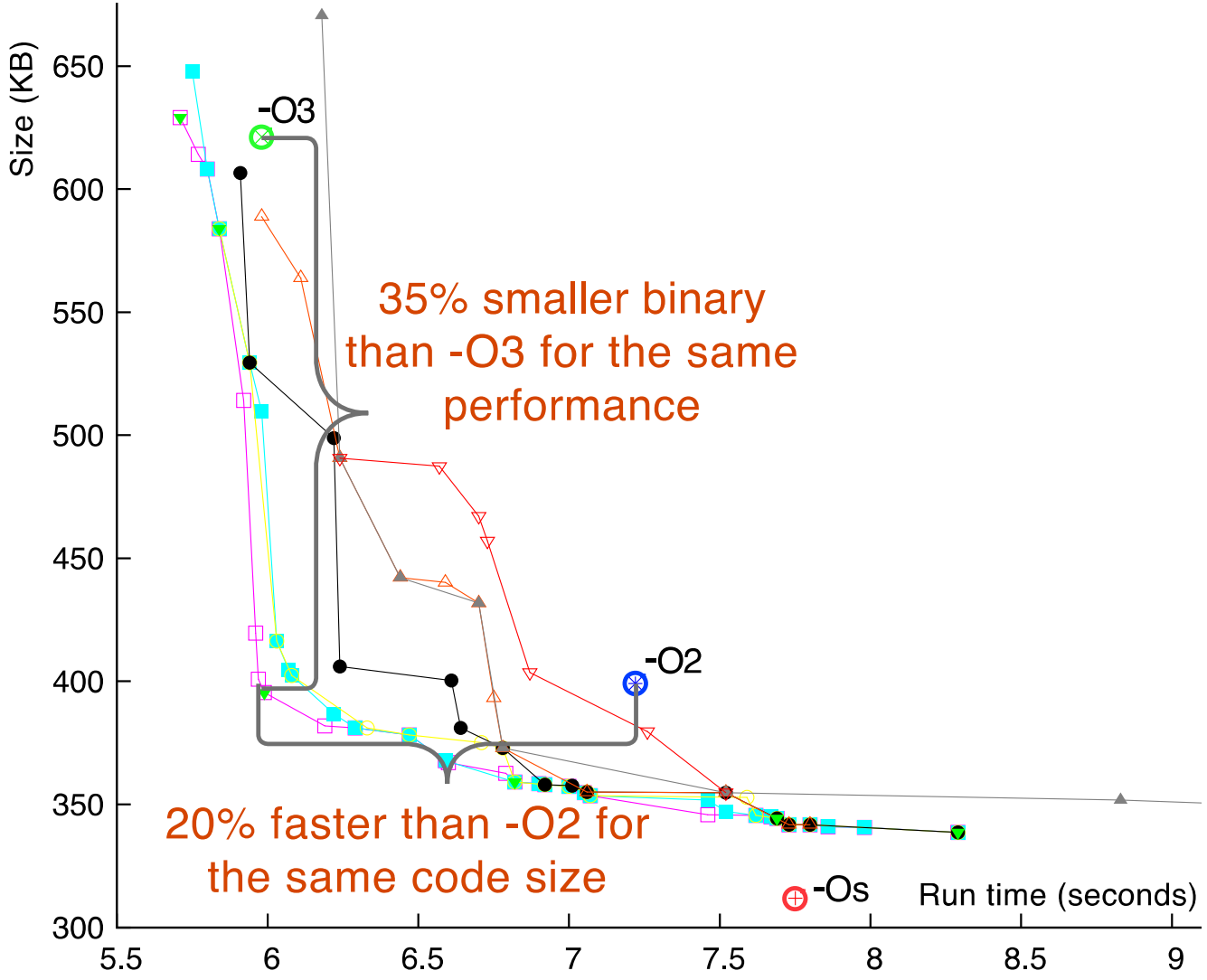
- There are few compiler options left for further manual analysis

Tuning SPEC2K INT (Thumb-2)



- We have tuned 6 of 12 SPEC2K INT tests on train data set
- The results were verified with ref dataset
- Half of the speedup persists across different data sets

Pareto Tuning for x264



Движки JavaScript (SFX, V8)

Динамические языки и VM

- **Динамические языки**
  - Основные особенности:
    - Управление памятью (сборка мусора, контроль доступа к объектам, границ при обращении к массивам, и т.п.)
    - Динамические типы (классы могут изменяться, а также создаваться новые во время выполнения)
    - Создание нового кода во время выполнения (*eval*)
  - Эти особенности делают статическую компиляцию затруднительной или невыгодной
  - Примеры: JavaScript, Python, Ruby (и в некоторой степени Java)

Интерпретация vs JIT компиляция

- Стандартный подход к реализации среды выполнения:
  - Программа компилируется в *байт-код* – набор простых инструкций, напоминающих ассемблер
  - Байт-код интерпретируется в виртуальной машине, которая также управляет динамическими объектами и предоставляет доступ к функциям времени выполнения

- JIT (Just-In-Time) Compilation – компиляция «на лету»:
  - Вместо интерпретации, байт-код сначала компилируется в код целевой архитектуры
  - Функции компилируются по мере необходимости, а не все заранее
  - Используется профилирование и спекулятивные оптимизации

JIT компиляция

- Преимущества JIT-компилятора:
  - Выполняет оптимизацию кода
  - Может использовать внутреннее представление более низкого уровня, и более подходящее для оптимизаций, чем байт-код (например, SSA)
  - Может оптимизировать программу под конкретные входные данные, т.к. работая во время выполнения, располагает данными о профиле программы («горячие» места, типы данных, значения переменных, вероятности переходов)
  - Если профиль изменился, может «на лету» перекомпилировать программу

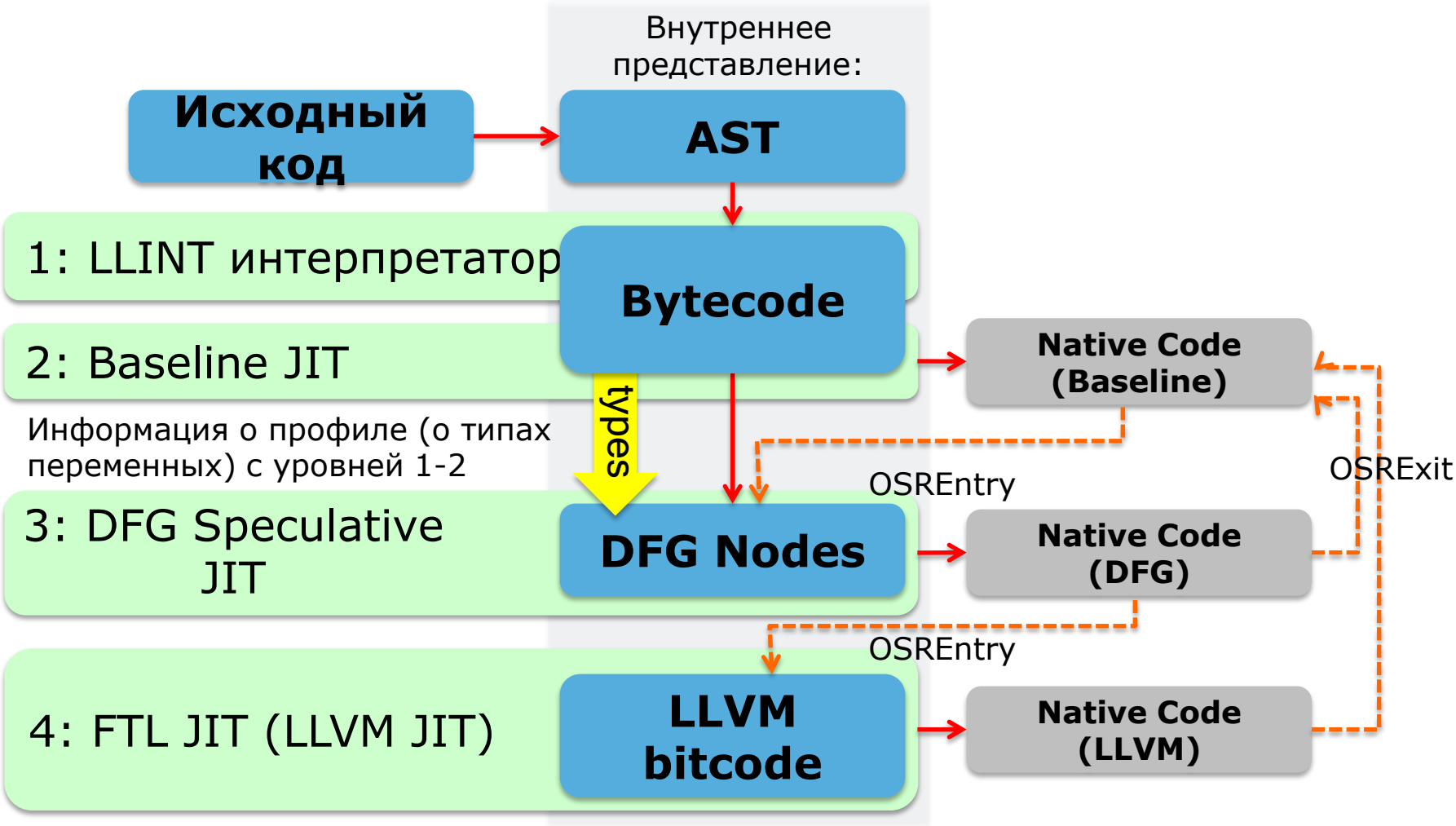
JIT компиляция

- Недостатки:
  - По сравнению с «обычным» (статическим) компилятором, сильно ограничен в сложности выполняемых оптимизаций, т.к. не должен задерживать выполнение программы
- Решение:
  - Оптимизировать только самые «горячие» места
  - Многоуровневый JIT: каждый уровень обрабатывает все более «горячие» места, сложность оптимизаций нарастает
  - Делать сложные оптимизации для следующего уровня параллельно с исполнением неоптимизированного кода на предыдущем уровне

Современные JavaScript-«движки»

- Самые популярные open-source проекты:
  - **SFX** (JavaScriptCore)
    - Используется в Safari и других браузерах на базе WebKit (BlackBerry)
    - Составная часть браузерного движка WebKit, разрабатывается Apple
  - **V8**
    - Используется в Google Chrome, встроенном браузере Android, а также Node.js
    - Основной JavaScript-движок в Blink (изначально был заменой для SFX в WebKit), разрабатывается Google
  - Mozilla SpiderMonkey
    - JS движок в Mozilla FireFox
- Общие особенности SFX и V8
  - Многоуровневый JIT, каждый уровень имеет собственное внутреннее представление и набор оптимизаций
  - Используется профилирование и спекулятивные оптимизации
  - Всего в ~2 раза медленнее C++ кода (SunSpider benchmark)

Устройство Webkit JavaScriptCore



Замена на стеке (On-Stack Replacement)

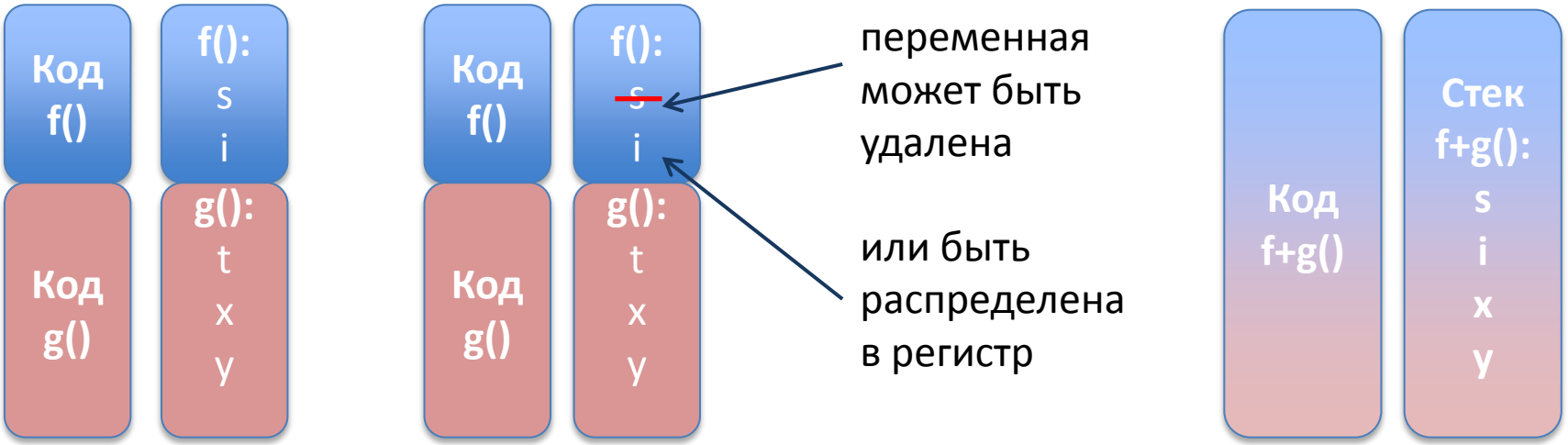
```
function g(t) {
  var x, y;
  …
}

function f(s) {
  for (var i = 0; i < 10000; i++) {
    s = g(i);
  }
}
```

i=2000 в DFG JIT
принято решение
о встраивании g() в
f():

i=1000: переход на
DFG JIT:

Baseline JIT:



По-разному оптимизированный код работает с разной раскладкой стека

Пример компиляции JavaScript в SFX JIT

JavaScript:

```
function foo() {
  var i, s;
  for (i = 0; i < 10000; i++) {
    s += i;
  }
  return s;
}
```

байткод LLINT / Baseline JIT:

```
[   0] enter
[   1] mov          r0, Int32: 0(@k0)
[   4] jnless           r0, Int32: 10000(@k1), 16(->20)
[   8] loop_hint
[   9] add          r1, r1, r0
[  14] pre_inc          r0
[  16] loop_if_less     r0, Int32: 10000(@k1), -8(->8)
[  20] ret          r1
```

Пример компиляции JavaScript в SFX JIT

11: < 1:2> GetLocal(@10, JS, r1(E), bc#9) predicting StringIntDoublerealDoublenanOther
0x7f86bc7bc8fe: mov 0x8(%r13), %rcx
13: < 2:3> GetLocal(@12, JS, r0(H<Int32>), bc#9) predicting Int
0x7f86bc7bc902: mov 0x0(%r13), %ebx
14: <!1:2>ValueAdd(@11, @13<Int32>, JS|MustGen|MightClobber|MayOverflow, bc#9)
0x7f86bc7bc906: or %r14, %rbx
0x7f86bc7bc909: mov %rcx, 0x10(%r13)
0x7f86bc7bc90d: mov %rbx, 0x18(%r13)
0x7f86bc7bc911: mov %rcx, %rsi
0x7f86bc7bc914: mov %rbx, %rdx
0x7f86bc7bc917: mov %r13, %rdi
0x7f86bc7bc91a: mov $0x9daff0, %r11
0x7f86bc7bc924: mov $0xbadbeef, (%r11)
0x7f86bc7bc92b: mov $0x9daff4, %r11
0x7f86bc7bc935: mov $0xbadbeef, (%r11)
0x7f86bc7bc93c: mov $0x0, -0x2c(%r13)
0x7f86bc7bc944: mov $0x7f8700738685, %r11
0x7f86bc7bc94e: call %r11
0x7f86bc7bc951: xor %esi, %esi
0x7f86bc7bc953: mov $0x9dc908, %r11
0x7f86bc7bc95d: mov (%r11), %r11
0x7f86bc7bc960: test %r11, %r11
0x7f86bc7bc963: jnz 0x7f86bc7bca05
15: < 2:-> SetLocal(@14, r1(E), bc#9) predicting StringIntDoublerealDoublenanOther
0x7f86bc7bc969: mov %rax, 0x8(%r13)

Пример компиляции JavaScript в SFX JIT

19: < 1:2> JSConstant(JS, $3 = Int32: 1, bc#14)
20: <!2:2>ArithAdd(@13<Int32>, @19<Int32>, Number|MustGen|CanExit, bc#14)
0x7f86bc7bc96d: mov 0x18(%r13), %rsi
0x7f86bc7bc971: mov %rsi, %rdi
0x7f86bc7bc974: add $0x1, %edi
0x7f86bc7bc977: jo 0x7f86bc7bca17
21: < 1:-> SetLocal(@20<Int32>, r0(H<Int32>), bc#14) predicting Int
0x7f86bc7bc97d: mov %edi, 0x0(%r13)
22: < 1:3> JSConstant(JS, $1 = Int32: 10000, bc#16)
23: <!1:3>CompareLess(@20<Int32>, @22<Int32>, Boolean|MustGen|MightClobber, bc#16)
0x7f86bc7bc981: cmp $0x2710, %edi
0x7f86bc7bc987: jl 0x7f86bc7bc8fe
24: <!0:-> Branch(@23<Boolean>, MustGen|CanExit, T:#1, F:#3, bc#16)

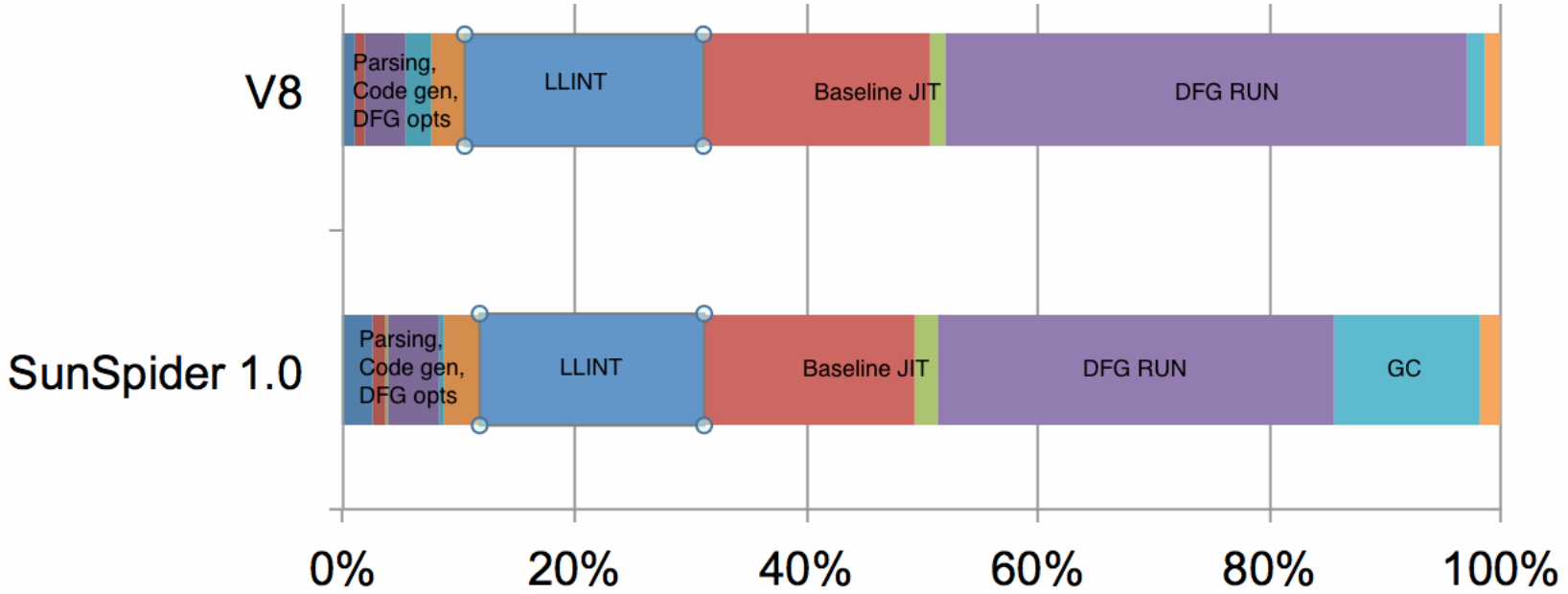
Особенности JIT

- Многоуровневый JIT
  - 1-й уровень: JIT с быстрой компиляцией (либо даже интерпретатор) – включает только самые простые оптимизации. Сразу же начинает выполнять байткод, и собирает информацию о профиле программы
  - 2-й уровень: оптимизирует только «горячие» места, выполняет более сложные оптимизации с использованием профиля, использует собственное внутреннее представление (например, SSA), может быть спекулятивным
  - Возможно большее число уровней, а также переключение между компиляторами разных уровней при изменении профиля

Особенности JIT

- Спекулятивный JIT
  - Специализирует код для данных, полученных при профилировании (прежде всего типы объектов). Для обработки остальных случаев предусмотрены проверки, возвращающие выполнение на предыдущий уровень JIT
  - Например, если профилирование показывает, что код работает только с целыми числами, можно использовать целочисленную арифметику и «обычные» регистры процессора, а результаты операций проверять на переполнение, и если оно произошло, продолжить выполнение на «базовом» JIT, код которого работает для любых типов объектов

Анализ работы JSC



- Построение AST
- Создание Bytecode
- Генерация BaselineJIT кода
- Компиляция DFG первый раз
- Перекомпиляция DFG
- SpeculativeJIT
- [Run] LL Interpreter
- [Run] BaselineJIT
- [Run] BaselineJIT after DFG
- [Run] DFG JIT
- Garbage Collection
- Miss

Компоненты JSC: сравнение

Время выполнения мульти-язычного теста PL benchmark от автора Martin Richards

Тест	Время, мс
Язык C	1.2
JavaScript обычный интерпретатор	129
LLINT интерпретатор	58
Baseline JIT	8.4
DFG JIT	2.1

Особенности оптимизации JavaScript

0 / -1 = ?

Особенности оптимизации JavaScript

0 / -1 = ?

1 / (0 / -1) = ?

Особенности оптимизации JavaScript

0 / -1 = -0

1 / (0 / -1) = -INF

Отрицательный ноль

```
function foo() {
  var x = 1;
  var y = -1;
  var s;
  var i;
  for (i = 0; i < 2001; i++) {
     if (i == 2000) x = 0;
     s = 1/(x/y);
  }
  return s;
}
var a = (1/(0/-1)).toString();
var b = foo().toString();
```

В выражении 1/(x/-1) переменная x меняет значение с 1 на 0.

Аналогичные примеры строятся для выражений

1/(-x) и 1/(x/-1+(-0)), а так же для выражения 1/(x % 4), в последнем x меняет значение с 1 на -4

Проверка на отрицательный ноль

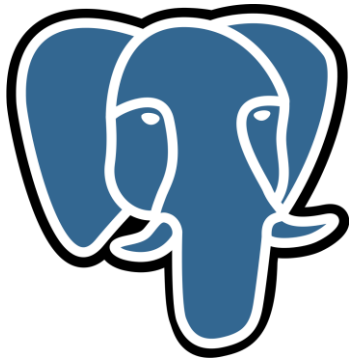
Результатом должно быть –Infinity
1 / (-0) ; 1 / (0 / -3) ; 1 / (-4 % 4)

Проверки при целочисленном делении
x % y равно -0, когда result == 0 и x < 0
x / y равно -0, когда x == 0 и y < 0

Проверка не всегда необходима:
5 / (A%B + 3)

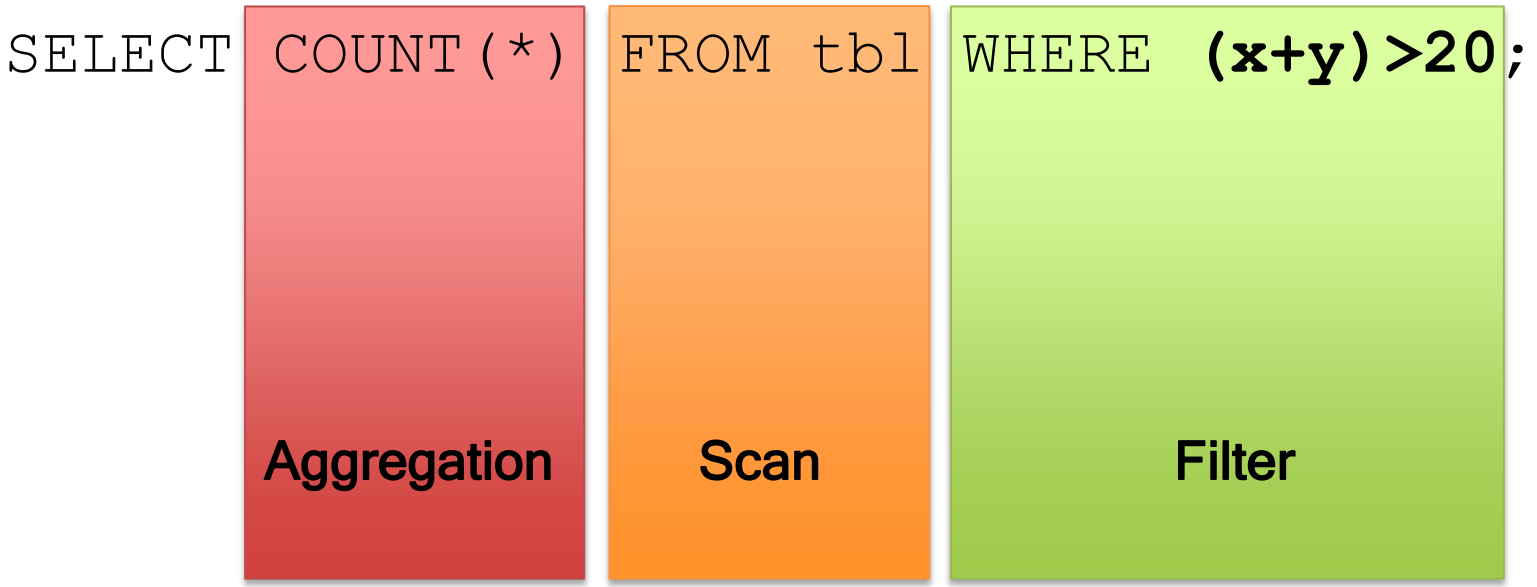
Ускорение исполнения запросов в PostgreSQL с использованием JIT-компилятора LLVM

Ускорение PostgreSQL

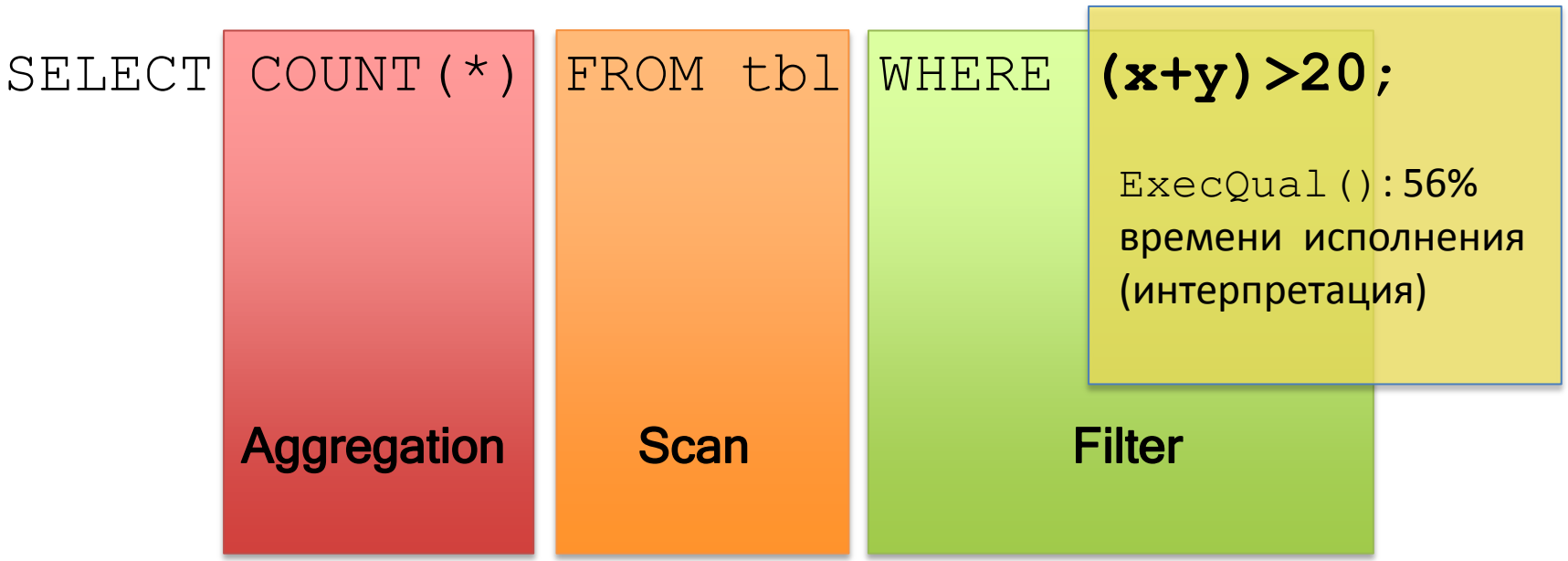


- Что именно мы хотим ускорить?
  - Сложные запросы, где «узким местом» в производительности является процессор, а не дисковые операции
    - OLAP, поддержка принятия решений, и т.д.
  - Цель: оптимизация производительности на наборе тестов TPC-H
- Как ускорить?
  - Использовать LLVM JIT, на первом этапе – для компиляции выражений в операторе WHERE

Пример оптимизации запроса



Пример оптимизации запроса



LLVM

- LLVM (Low Level Virtual Machine) – компиляторная инфраструктура для компиляции и оптимизации программ
  - Платформенно-независимое внутреннее представление (LLVM bitcode)
  - Широкий набор оптимизаций
  - Кодогенерация под популярные платформы (x86, x86_64, ARM, MIPS, …)
  - Подходит для построения JIT-компиляторов: динамическая библиотека с API для генерации биткода, оптимизации и кодогенерации
  - Лицензия: UIUC (permissive BSD-like)
  - Сравнительно простой код, легко разобраться

Использование LLVM JIT – популярный тренд

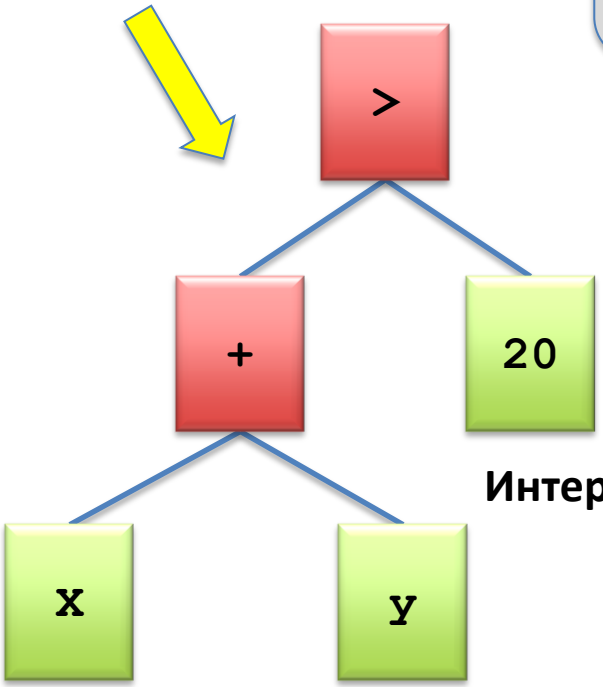
- Pyston (Python, Dropbox)
- HHVM (PHP & Hack, Facebook)
- LLILC (MSIL, .NET Foundation)
- Julia (Julia, community)

- JavaScript:
  - JavaScriptCore в WebKit (JavaScript, Apple)
  - LLV8 – добавление LLVM в качестве дополнительного уровня JIT в Google V8 (JavaScript, ISP RAS)

Что получится, если добавить к Postgres LLVM JIT?

Вычисление выражений

(x+y)>20



Генератор LLVM-биткода

Биткод LLVM

```
define i32 @where_expr(i32 %x, i32 %y) #0 {
entry:
  %add = add nsw i32 %y, %x
  %cmp = icmp sgt i32 %add, 20
  %conv = zext i1 %cmp to i32
  ret i32 %conv
}
```

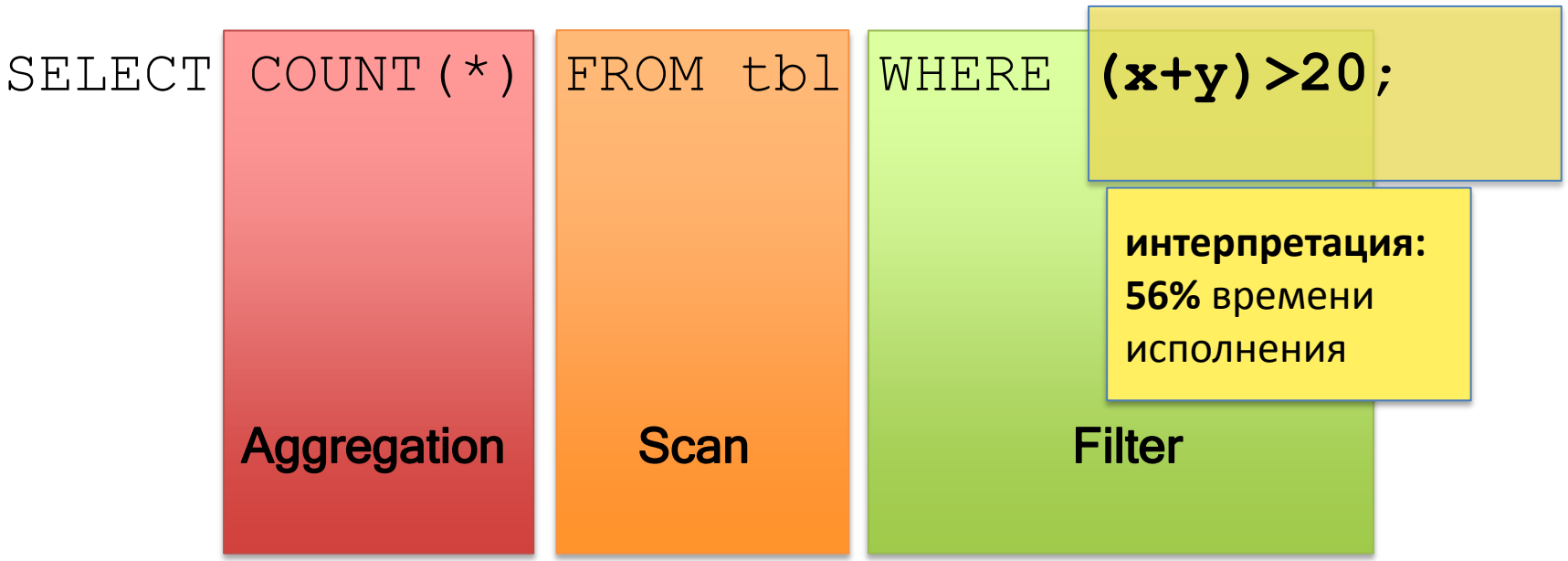
LLVM MCJIT

Интерпретация vs **Оптимизированный двоичный код**

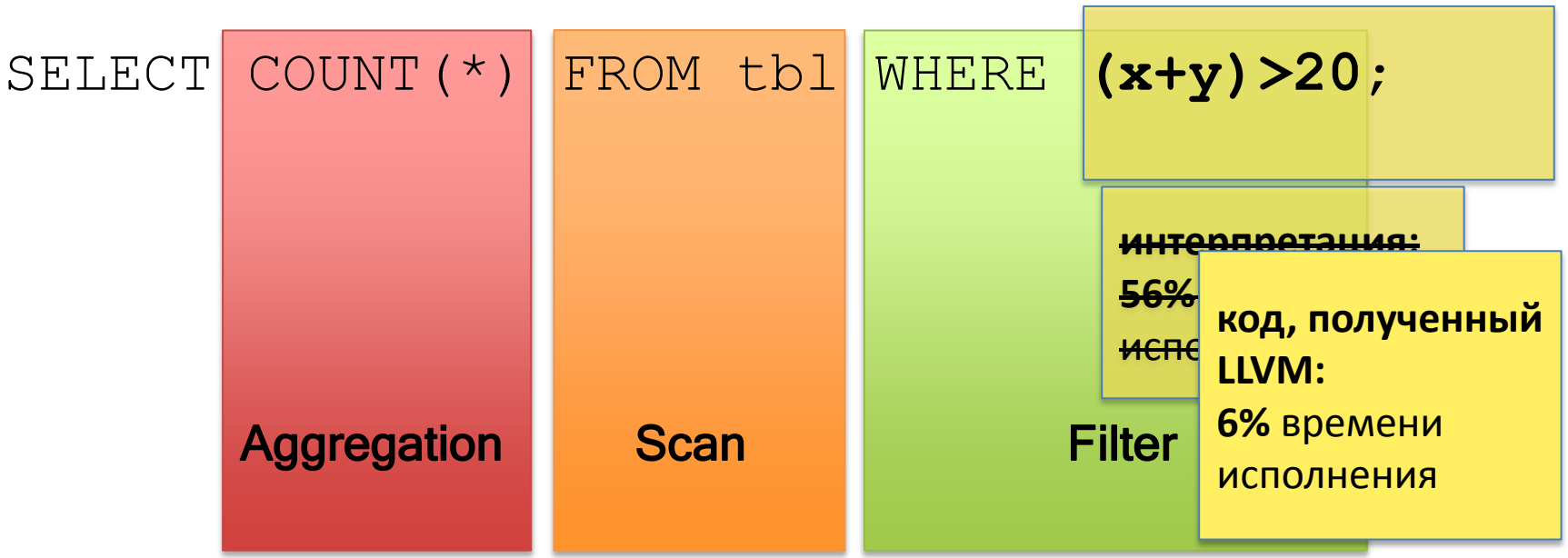
```
foo:
        addl %esi, %edi
        cmpl $20, %edi
        setg %al
        movzbl    %al, %eax
        retq
```

в ~10 раз быстрее

Пример оптимизации запроса



Пример оптимизации запроса



=> Ускорение выполнения запроса в 2 раза

Related work (1)

1. T. Neuman. Efficiently Compiling Efficient Query Plans for Modern Hardware. Proceedings of the VLDB Endowment, Vol. 4, No. 9, 2011.
2. PGStrom
   - Расширение для PostgreSQL (Custom scan)
   - Выполнение запросов на GPU (Cuda)
3. Axle Project
   - OpenCL

Related work (2)

4. Vitesse DB
   - Коммерческое, проприетарное ПО на основе PostgreSQL
   - Реализована часть идей из [1]:
     - Aggregation, Scan и Filter выполнены на LLVM в виде одной функции
     - Компиляция выражений в операторе WHERE с помощью LLVM
   - Ускорение в 2-8 раз на Q1, в ~3 раза в среднем на TPC-H

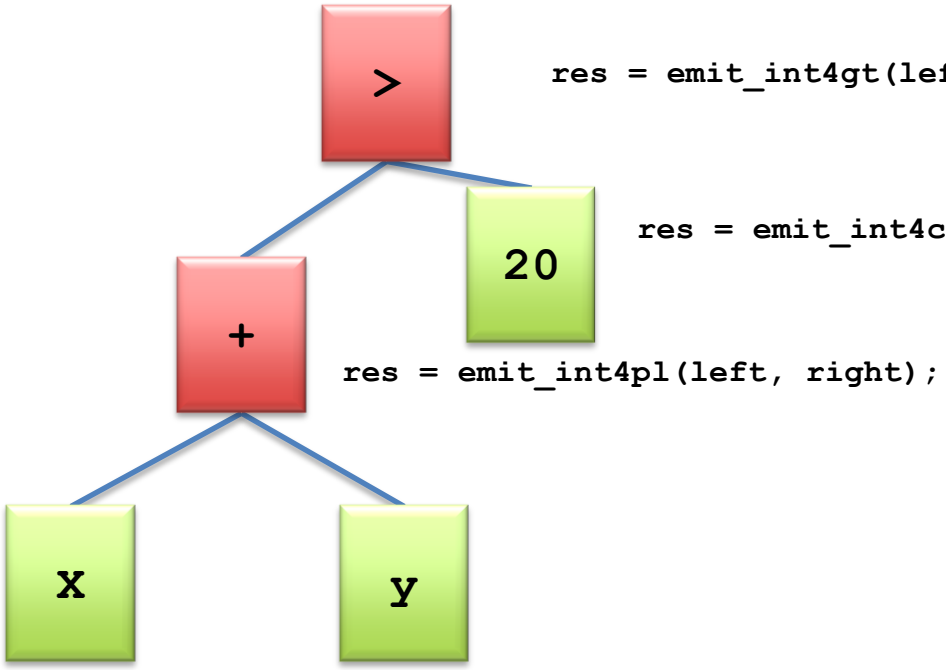
1-й шаг: компиляция выражений

- Расширение PostgreSQL (Custom Scan)
  - Для SeqScan изменена фильтрация:
    - Добавлена компиляция дерева выражений в LLVM
    - Исполнение оптимизированного кода для фильтрации каждого tuple
    - Поддерживаются базовые операции для int, float, Date (около 100 из ~2000)
- Результат: ускорение в 2 раза для простых синтетических тестов
  - удаление неявных вызовов для каждой операции
  - константы в запросе => значения в инструкции
  - оптимизация выражений средствами LLVM

Поддержка операций в LLVM

- При обходе дерева выражений операция в узле не выполняется, а создает LLVM-биткод, который ей соответствует
- Необходимо реализовать кодогенерацию для всех операций и всех типов, поддерживаемых в выражениях (около 2000)
  1. Реализация всех операций вручную
     - однообразная работа, легко допустить ошибку, сложно поддерживать
  2. Предварительная компиляция кода операций в LLVM-биткод: *src/backend/*.c* => *.bc => all_ops.bc
     - Долгая компоновка
  3. Автоматическая компиляция операций *src/backend/*.c* в Си-код, использующий LLVM API для генерации кода из п.1
     - Соответствующий инструмент из LLVM устарел и не поддерживается

Кодогенерация для выражений




```
res = emit_int4gt(left, right);
```



```
res = emit_int4const(20);
```



```
res = emit_int4pl(left, right);
```



```
res = emit_load_int4_var('X');
```

```
res = emit_load_int4_var('Y');
```

1. *emit_int4gt() { LLVMBuildICmp(…); }*
2. *LLVMBuildCall("int4gt");*
   код int4gt уже скомпилирован при сборке PostgreSQL и подгружен из .bc-файла. При компиляции вызов заменится на тело функции.
3. *emit_int4gt()* как в п.1, но был создан автоматическим генератором на этапе сборки, как в п.2

Профилирование TPC-H

TPC-H Q1:

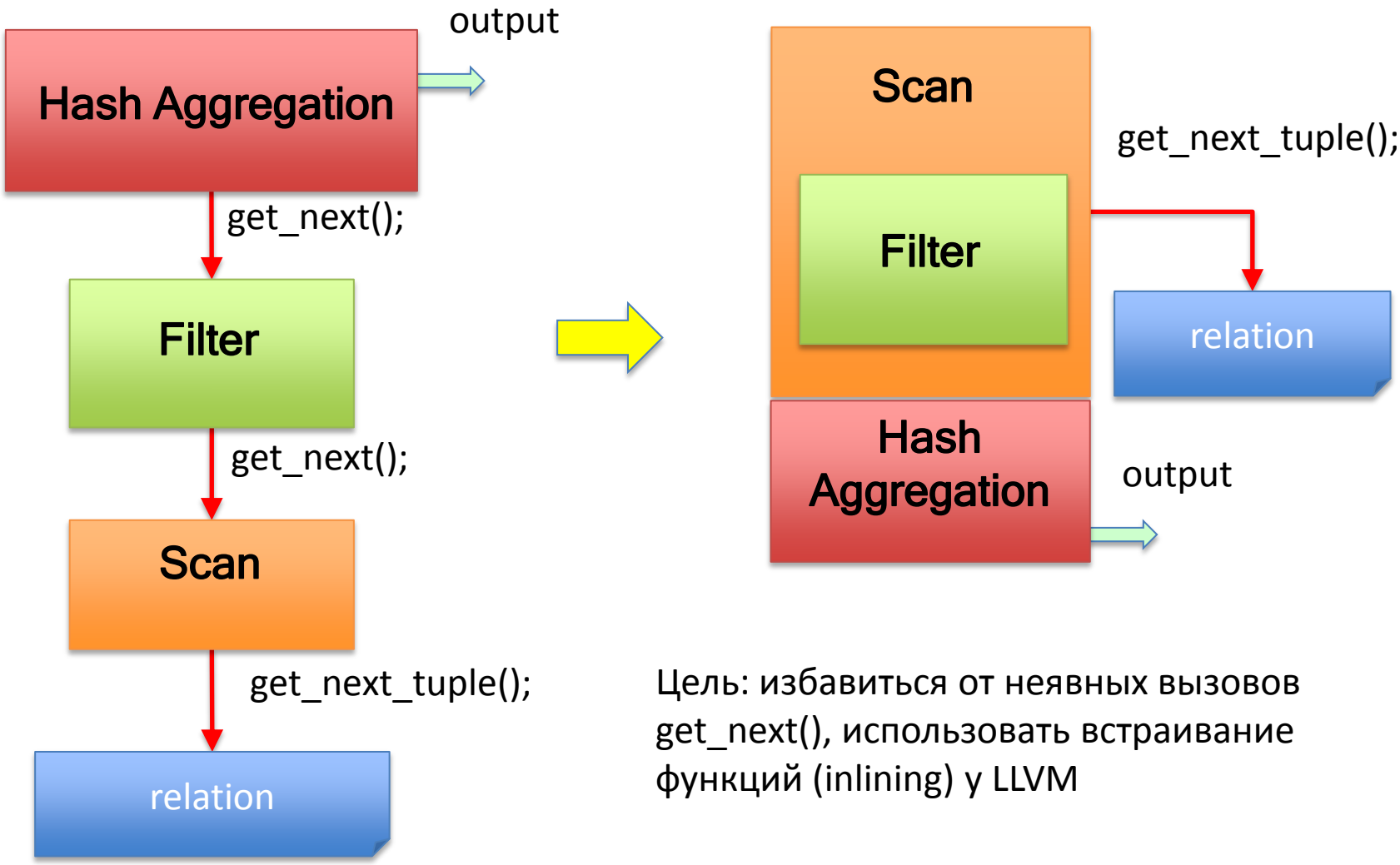
```
select
      l_returnflag,
      l_linestatus,
      sum(l_quantity) as sum_qty,
      sum(l_extendedprice) as sum_base_price,
      sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
      sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
      avg(l_quantity) as avg_qty,
      avg(l_extendedprice) as avg_price,
      avg(l_discount) as avg_disc,
      count(*) as count_order
from
      lineitem
where
      l_shipdate <=
      date '1998-12-01' -
      interval '60 days'
group by
      l_returnflag,
      l_linestatus
order by
      l_returnflag,
      l_linestatus;
```

Function	TPC-H Q1	TPC-H Q2	TPC-H Q3	TPC-H Q6	TPC-H Q22	Average on TPC-H
ExecQual	6%	14%	32%	3%	72%	25%
ExecAgg	75%	-	1%	1%	2%	16%
SeqNext	6%	1%	33%	-	13%	17%
IndexNext	-	57%	-	-	19%	38%
BitmapHeapNext	-	-	-	85%	-	85%

2-й шаг: реализация Scan и Aggregation на LLVM

- Расширение PostgreSQL (Executor hook)
  - Реализация Scan, Aggregation на LLVM
    - Поддерживаются SeqScan, IndexScan, IndexOnlyScan, DirectAggregation, HashAggregation
  - Отказ от “Volcano-Style” итерационной модели
    - Реализация HashAggregation, SeqScan и Filter на LLVM в виде одной функции, внешний цикл выполняет Scan
- Результат: на данный момент ускорение ~25% на TPC-H Q1

Избавление от итерационной “Volcano-style” модели



Цель: избавиться от неявных вызовов get_next(), использовать встраивание функций (inlining) у LLVM

Заключение

- Разработано расширение PostgreSQL для динамической компиляции SQL-запросов с помощью LLVM JIT. Реализованы фазы:
  - Фильтрация (компилируются выражения из оператора WHERE, поддерживаются типы int, float, Date, Numeric)
  - Сканирование (SeqScan, IndexScan, IndexOnlyScan)
  - Агрегация (DirectAggregation, HashAggregation; sum, avg, count)
- Результаты:
  - Ускорение в ~2 раза на простых синтетических тестах
  - Ускорение на ~25% на TPC-H Q1

Будущая работа

- Компиляция выражений: реализовать все операции и типы данных на LLVM, или скомпилировать автоматически из кода Postgres
- Реализовать на LLVM все виды сканирования и агрегации, сортировки (в т.ч. в одном цикле)
- Реализовать поддержку JOIN и подзапросов
- Реализовать на LLVM *slot_deform_tuple()* под конкретный запрос
- Тестирование на всех тестах TPC-H и других бенчмарках, профилирование, поиск мест для оптимизации

Будущая работа

- Использование параллелизма:
  - Реализация параллельного сканирования и агрегации на LLVM
  - Параллельная компиляция
- Подготовка к релизу в Open Source, взаимодействие с PostgreSQL Community

Спасибо!

Вопросы, пожелания, обратная связь:
dm@ispras.ru