

Лекция 3

Оптимизация ветвлений и размещения кода

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Параллельные вычислительные технологии»

Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Ветвления в программах C/C++

```
if (x == 0)
else if (x == 1)
else
```

```
switch (x) {
    case 0:
        break;
    case 1:
        break; default:
}
}
```

```
for (i = 0; i < 10; i++) {
}
```

```
while (data > 0) {
    data--;
}
```

```
do {
    data--;
} while (data > 0);
```

Ветвления и базовые блоки

```
int a = 1, b = 3, c = 0;

if (a < b) {
    c = 4; // basic block
}
```



```
movl    $1, -8(%rbp)    # a
movl    $3, -4(%rbp)    # b
movl    $0, -12(%rbp)   # c
```

```
movl    -8(%rbp), %eax
cmpl  -4(%rbp), %eax   # compare a, b
jge  .L2              # if a >= b goto .L2 (Jump If Greater or Equal)
movl    $4, -12(%rbp)   # else: c = 4 (basic block)
.L2:
...
```

- **Базовый блок** (basic block) – последовательность инструкций с единственным входом и единственным выходом (jmp, jXX, call, ret)
- Базовый блок может содержать переход только в конце, каждая инструкция блока выполняется один раз
- Базовые блоки – это узлы графа потока управления (control flow graph – CFG)

Ветвления в программах (gcc 11.2)

```
if (a > 2) {  
    printf("A\n");  
} else {  
    printf("B\n");  
}
```

```
$ gcc ./prog.c --save-temps
```

```
    cmpl    $2, -4(%rbp) # compare a, 2  
    jle    .L5          # jmp if a <= 2  
# a > 2, print A  
    leaq   .LC0(%rip), %rax  
    movq   %rax, %rdi  
    call   puts@PLT  
    jmp   .L6  
.L5:  
# a <= 2, print B  
    leaq   .LC1(%rip), %rax  
    movq   %rax, %rdi  
    call   puts@PLT  
.L6:
```

```
$ gcc -O2 ./prog.c --save-temps
```


```
    cmpl    $2, %edi    # compare a, 2  
    jle    .L4          # jmp if a <= 2  
# a > 2  
    leaq   .LC0(%rip), %rdi  
    call   puts@PLT  
.L5:  
    xorl   %eax, %eax  
    addq   $8, %rsp  
    ret  
.L4:  
# a <= 2  
    leaq   .LC1(%rip), %rdi  
    call   puts@PLT  
    jmp   .L5
```

Ветвления в программах (gcc 11.2)

```
for (int i = 0; i < N; i++) {  
    printf("%d\n", i);  
}
```

```
$ gcc -O2 ./prog.c --save-temps
```

```
    testl    %edi, %edi          # N & N  
    jle     .L6                 # jmp if N==0  
    # ...  
    movl    %edi, %ebp  
    xorl    %ebx, %ebx         # i = 0  
    .L3:  
    movl    %ebx, %edx  
    movq    %r12, %rsi  
    movl    $1, %edi  
    xorl    %eax, %eax  
    call    __printf_chk@PLT  
    addl    $1, %ebx           # i = i + 1  
    cmpl    %ebx, %ebp        # cmp N, i  
    jne     .L3                 # jmp if N != i  
    .L6:
```



Ветвления в программах (gcc 11.2)

```
switch (a) {  
  case 1:  
    printf("1\n");  
    break;  
  case 2:  
    printf("2\n");  
    break;  
  case 3:  
  case 4:  
    printf("3-4\n");  
    break;  
  default:  
    printf("default\n");  
}
```

```
$ gcc -O2 ./prog.c --save-temps
```

```
.LFB23:  
  cmpl    $2, %edi          # compare a, 2  
  je     .L2                # jmp if a = 2  
  jg     .L3                # jmp if a > 2  
  cmpl    $1, %edi  
  jne    .L5  
  leaq   .LC0(%rip), %rdi   # a=1  
  call   puts@PLT  
.L7:  
  xorl   %eax, %eax        # return  
  addq   $8, %rsp  
  ret  
.L3:  
  subl   $3, %edi  
  cmpl   $1, %edi  
  ja     .L5  
  leaq   .LC2(%rip), %rdi   # a=3  
  call   puts@PLT  
  jmp    .L7  
.L2:  
  leaq   .LC1(%rip), %rdi   # a=2  
  call   puts@PLT  
  jmp    .L7  
.L5:  
  leaq   .LC3(%rip), %rdi   # a>=4  
  call   puts@PLT  
  jmp    .L7
```

Конфликты управления (Control hazards)

Этапы выполнения инструкций на конвейере

```
movl %ebx, %eax
cmpl $10, %eax      # eax == 10?
jne  not_equal     # jmp if eax != 10
# eax == 10
movl %eax, %ecx
jmp end

not_equal:
# %eax != 10
movl $-0x1, %ecx

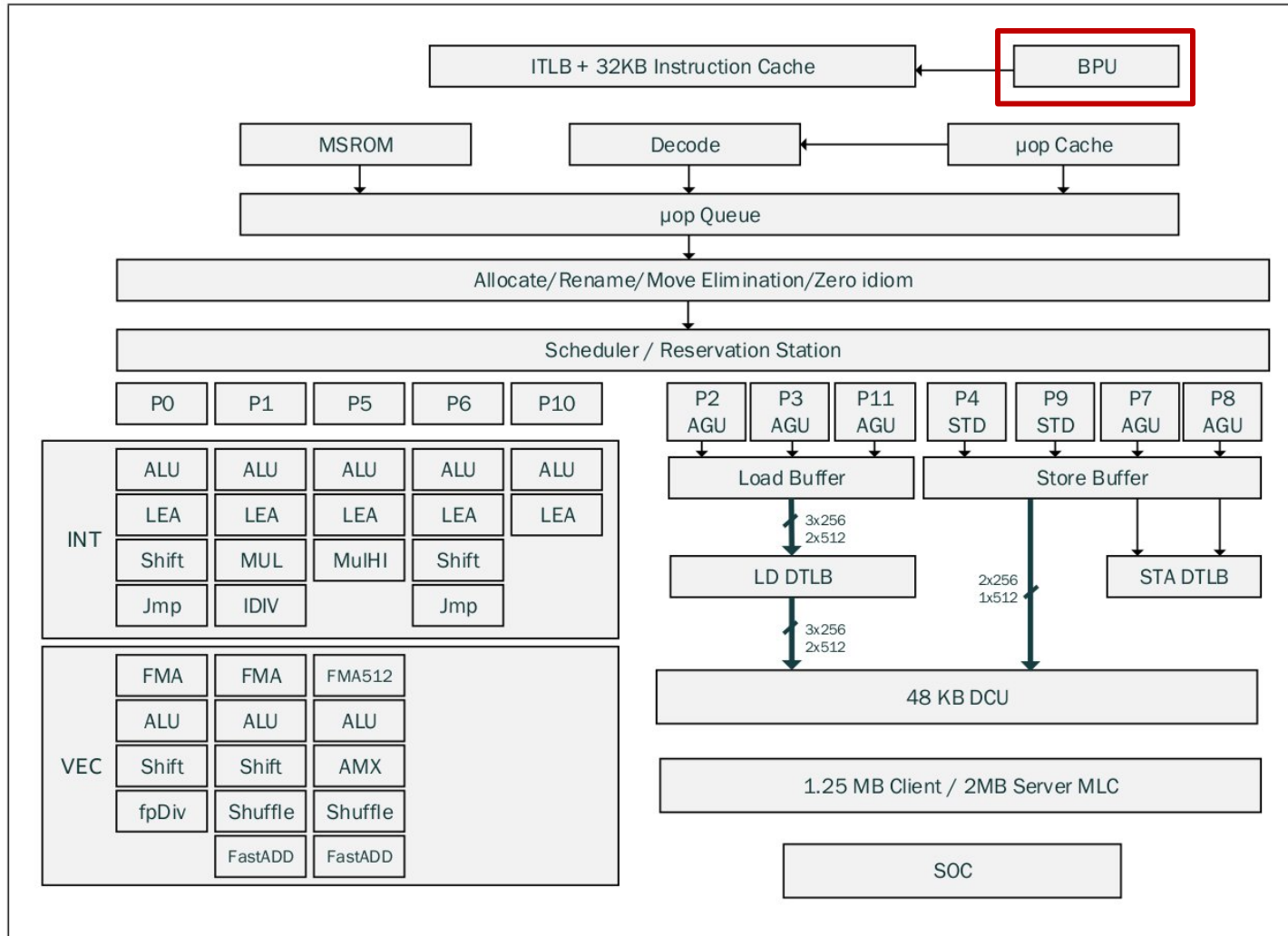
end:
```

Step	IF	ID	EX	ST
1	movl			
2	cmpl	movl		
3	jne	cmpl	movl	
4	???	jne	cmpl	movl
5				
6				
7				

Какую инструкцию выбрать из памяти (IF) на шаге 4?
Инструкция `cmpl` еще не выполнена, а `jne` декодируется (ID)

В процессоре присутствует **модуль предсказания переходов** (Branch Prediction Unit – BPU), который управляет счетчиком команд

Intel 64 (Golden Cove uarch, 2021)



- **BPU** (Branch Prediction Unit)

Оптимизация кода для эффективной загрузки CPU Frontend (IF, ID)

- **Минимизация количества условных переходов** (меньше записей в BTB)
 - Раскрутка циклов, инструкции cmov, setcc, SIMD masked operations
- **Минимизация промахов при доступе к кеш-памяти инструкций**
- **Intel 64 and IA-32 Architectures Optimization Reference Manual**
 - Код и данные следует размещать на разных страницах памяти
 - Организация кода в соответствии с логикой работы алгоритма статического предсказания переходов
 - Раскручивание многократно выполняемых циклов (unroll), так чтобы в них оставалось 16 или менее итераций
 - Избегать множественных условных переходов на общий блок кода, выровненный на границу в 8 байт (ограничение устранено начиная с Ice Lake)

Предсказание переходов (branch prediction)

- **Модуль предсказания условных переходов** (Branch Prediction Unit, BPU) – модуль процессора, определяющий по адресу инструкции ветвления *будет ли выполнен переход и по какому адресу*
- Предсказывает условные переходы, вызовы/возвраты из функций
- Вероятность правильного предсказания переходов в современных процессорах превышает 0.9
- После предсказания процессор начинает спекулятивно выполнять инструкции с предсказанного адреса (speculative execution)
- **Альтернативный подход (без BPU)** – спекулятивно выполнять обе ветви ветвления, пока не будет вычислено управляющее выражение (условие)

```
    cmpl    $2, -4(%rbp) # cmp a,2
    ? jle   .L5          # jmp if a<=2
    # a > 2, print A
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    call    puts@PLT
    jmp     .L6
.L5:
    # a <= 2, print B
    leaq    .LC1(%rip), %rax
    movq    %rax, %rdi
    call    puts@PLT
.L6:
```

Статическое предсказание переходов (static prediction)

- Косвенные переходы
- **Section 3.4.1 Branch Prediction Optimization [*]**
 - Predict forward conditional branches to be NOT taken
 - Predict backward conditional branches to be taken
 - Predict indirect branches to be NOT taken (switch, call funcptr)

```
Begin:  mov    eax, mem32
        and    eax, ebx
        imul  eax, edx
        shld  eax, 7
        jc    Begin
```

Статическое предсказание:
обратный переход
выполняется

```
        mov    eax, mem32
        and    eax, ebx
        imul  eax, edx
        shld  eax, 7
        jc    Begin
        mov    eax, 0
Begin:  call   Convert
```

Статическое предсказание:
прямой переход не выполняется
(fall-through)

```
//Forward condition branches not taken (fall through)
IF<condition> {...
  ↓
}

IF<condition> {...
  ↓
}

//Backward conditional branches are taken
LOOP {...
  ↑ — }<condition>

//Unconditional branches taken
JMP
----->
```

Наиболее
вероятный блок
следует размещать
сразу после if

Статическое предсказание переходов (static prediction)

- **Косвенные переходы** (indirect branches) могут иметь несколько целевых адресов, по которым осуществляются переходы
 - `jmp rax` – переход по адресу в регистре `rax`
 - адрес в регистре `rax` может меняться на каждой итерации цикла
- В BTB (Branch Target Buffer) с адресом инструкции перехода связан один целевой адрес, что осложняет прогнозирование косвенных переходов

Динамическое предсказание переходов

- **Динамическое предсказание основано на** хранении и анализе истории условных переходов
- **BTB (Branch Target Buffer)** – ассоциативный массив (хеш-таблица), сопоставляющий адресу инструкции ветвления историю переходов и целевой адрес перехода (target)

```
0xFF01 l1:  movl %ebx, %eax
0xFF02      cmpl $0x10, %eax
0xFF03      jne  l2
0xFF04      movl %eax, %ecx
0xFF05      jmp  l3
0xFF06 l2:  movl $-0x1, %ecx
0xFF07 l3:
0xFF08      cmpl $0xFF, %ebx
0xFF09      jne  l1
```

Instr. Address (low bits)	History	Target Address
0xFF03	1	0xFF06
0xFF09	0	

Динамическое предсказание переходов

- На этапе выборке инструкции условного перехода происходит обращение в БТВ:
 - Если запись для адреса инструкции перехода (%IP) присутствует в БТВ, значит сохранены история и целевой адрес перехода (target address)
 - На основе истории ветвлений строится прогноз:
 - осуществлять переход (to be taken) – БТВ возвращает target address
 - не осуществлять переход (not to be taken) – БТВ возвращает %IP + <instr-width>

```
0xFF01 l1:  movl %ebx, %eax
0xFF02      cmpl $0x10, %eax
0xFF03      jne  l2
0xFF04      movl %eax, %ecx
0xFF05      jmp  l3
0xFF06 l2:  movl $-0x1, %ecx
0xFF07 l3:
0xFF08      cmpl $0xFF, %ebx
0xFF09      jne  l1
```

Instr. Address (low bits)	History	Target Address
0xFF03	1	0xFF06
0xFF09	0	

1-bit dynamic predictor

Адрес инструкции перехода (IP):



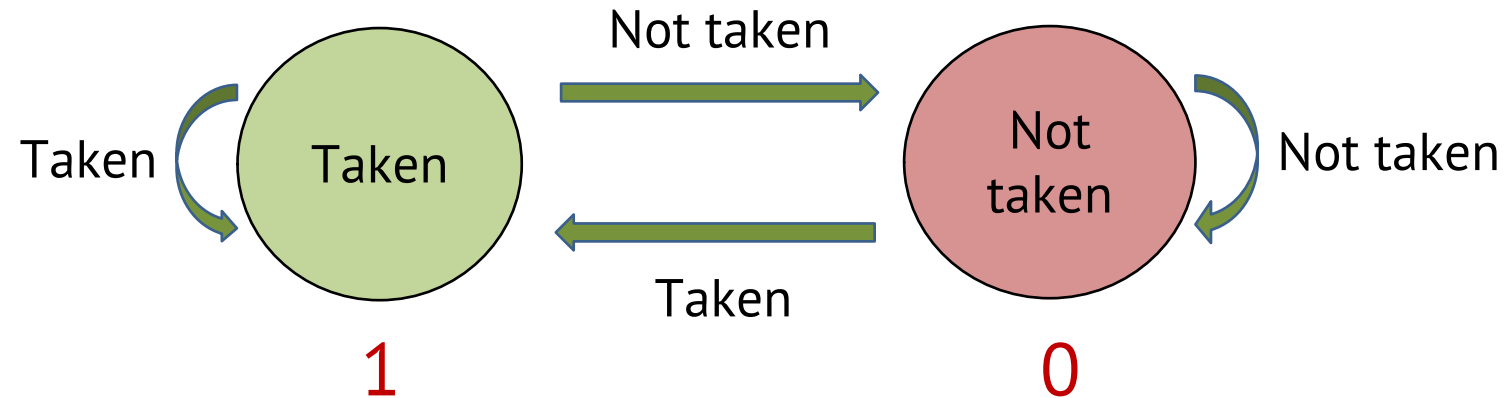
Branch Target Buffer (BTB)

Record	Branch History	Target
0	1	0xAF06
1	0	0x1134
2	1	0x01FC
3	0	0xFF06
...		
$2^k - 1$	1	0xBEAF

Branch History (1 bit)

- **0** – ветвление не состоялось, не осуществлять переход
- **1** – ветвление состоялось, осуществлять переход
- Неправильный прогноз – отмена операций по ложной ветви
- История корректируется после выполнения операции сравнения (cmp)

1-bit dynamic predictor



```
for (i = 0; i < 6; i++) {  
    if ((i % 2) == 0)  
        /* Even */  
    else  
        /* Odd */  
}
```

Точность 0%

Iter, i	Predicted	Real
0	0 (NOT TAKEN)	1 (TAKEN)
1	1	0
2	0	1
3	1	0
4	0	1
5	1	0

Saturating 2-bit counter (bimodal predictor)

- **Корректировка счетчика (после выполнения сравнения)**

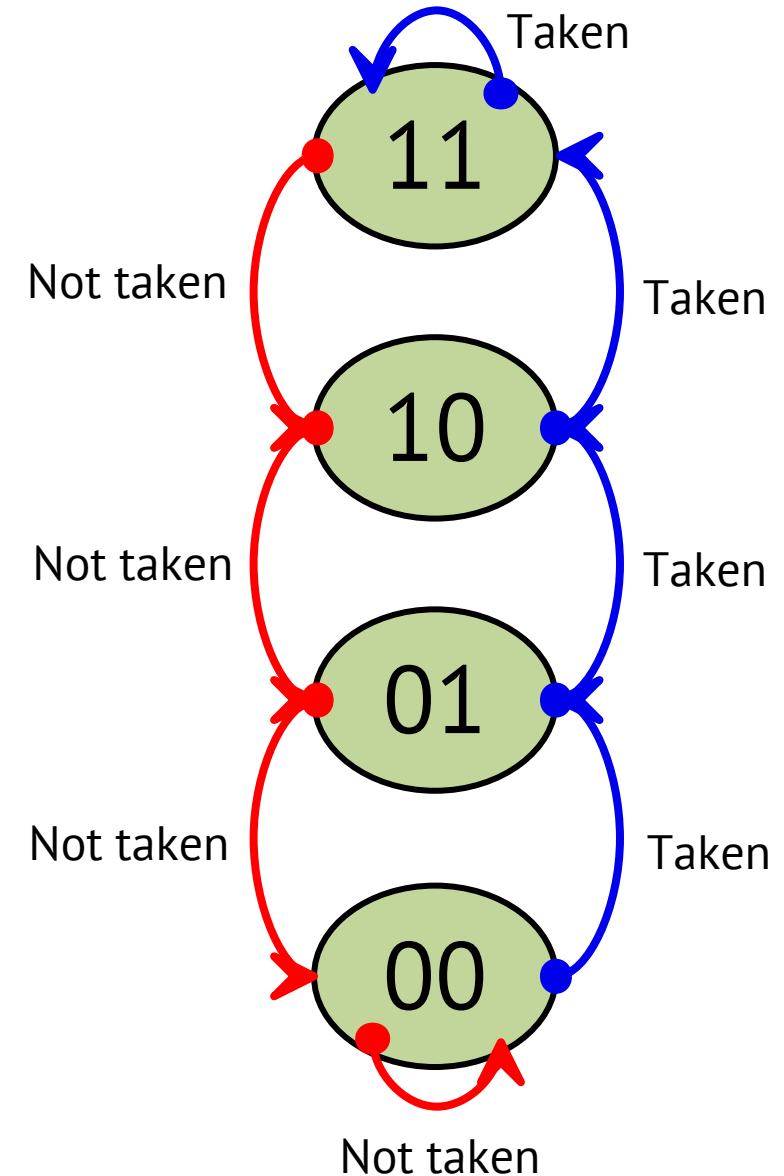
- переход выполнен: счетчик увеличивается на 1
 $\text{counter} = \min(3, \text{counter} + 1)$
- переход не выполнен: счетчик уменьшается на 1
 $\text{counter} = \max(0, \text{counter} - 1)$

- **Предсказание:**

- $\text{counter} < 2$: переход не выполняется
- $\text{counter} \geq 2$: переход выполняется

- Использовался в Intel Pentium

- n -битный предсказатель
если $\text{counter} \geq 2^n / 2$, то ветвление выполняется



Реализации ВТВ

- **Intel Pentium:** saturating 2-bit counter
- **Intel Pentium {MMX, Pro, II, III}:**
two-level adaptive branch predictor (4-bit history)
- **Pentium 4:** Agree predictor (16-bit global history)
- **Intel Atom:** two-level adaptive branch predictor
- **Intel Nehalem:** two-level branch predictor, misprediction penalty is at least 17 clock cycles

Agner Fog. **The microarchitecture of Intel, AMD and VIA CPUs**
(an optimization guide for assembly programmers and compiler makers)

<http://www.agner.org/optimize/microarchitecture.pdf>

Размещение блоков кода с учетом статического предсказания

- Наиболее вероятный блок следует размещать сразу после if
 - Согласуется с алгоритмом статического предсказания
 - Учитывая возможную аппаратную предвыборку инструкций из памяти (instruction cache prefetching)

```
p = malloc(size);
if (p == NULL) {
    error();
    break;
} else {
    process(p);
    free(p);
}
```



```
p = malloc(size);
if (p != NULL) {
    process(p);
    free(p);
} else {
    error();
    break;
}
```

Predict forward conditional branches
to be NOT taken (fall-through)

Устранение ветвлений (branchless code)

Устранение ветвлений (branchless code)

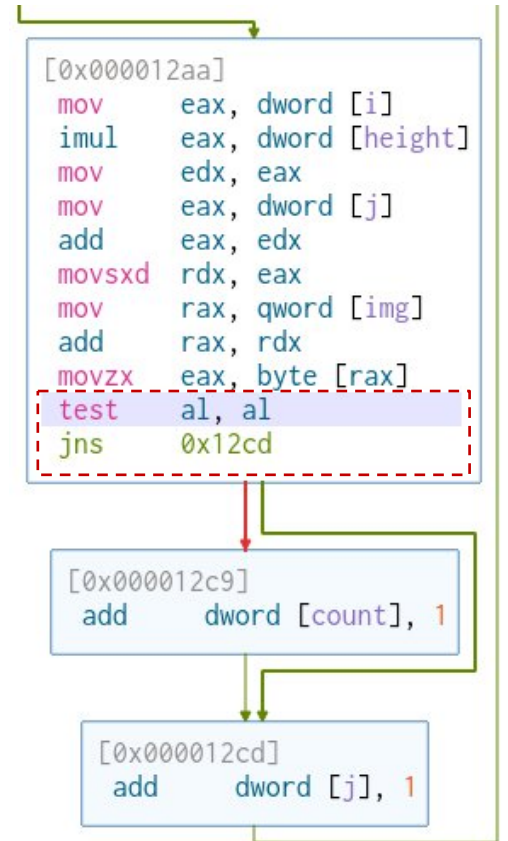
- Вынос инвариантных ветвлений за цикл
- Замена ветвлений арифметическими выражениями
- Устранение ветвлений с использованием операций условной установки и копирования (setcc, movcc)
- Устранение ветвлений с использованием векторных операций — маскирование
- Раскрутка циклов

Устранение ветвлений: сравнение со скаляром (`expr >= scalar`)

```
enum {W = 15360, H = 8640};

int image_is_dark(uint8_t *img, int width, int height)
{
    int count = 0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (img[i * width + j] >= 128) { // test + jns (SF=0)
                count++;
            }
        }
    }
    return count < width * height / 2;
}
```

- Ветвление JNS на каждой итерации цикла ($\text{width} * \text{height}$ условных переходов)
- Цель – устранить ветвление для минимизации ошибок предсказания переходов



```
$ gcc -g -o prog ./prog.c
$ perf stat -e branch-misses -- taskset --cpu-list 0 ./prog
Time 0.747575, dark 0
```

```
Performance counter stats for 'taskset --cpu-list 0 ./prog':
    70721638      branch-misses
```

Устранение ветвлений: сравнение со скаляром (shr)

```
int image_is_dark_v2(uint8_t *img, int width, int height)
{
    int count = 0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            count += (img[i * width + j] >> 7); // shr al, 7; деление на 27
        }
    }
    return count < width * height / 2;
}
```

- **Условное выражение заменено на арифметическое**
- Значение типа пикселя `img[i][j]` в диапазоне `[0, 255]`, поэтому целочисленное деление на 128 дает 0 либо 1 (округление вниз, floor)

```
counter += img[i * width + j] / 128
```

```
[0x0000132b]
mov     eax, dword [i]
imul   eax, dword [height]
mov     edx, eax
mov     eax, dword [j]
add     eax, edx
movsxd rdx, eax
mov     rax, qword [img]
add     rax, rdx
movzx   eax, byte [rax]
shr     al, 7
movzx   eax, al
add     dword [count], eax
add     dword [j], 1
```

```
$ perf stat -e branch-misses -- taskset --cpu-list 0 ./prog
Time 0.280183, dark 0
```

```
Performance counter stats for 'taskset --cpu-list 0 ./prog':
```

```
4364970      branch-misses
```

```
# Ускорение 2.7
```

Устранение ветвлений: сравнение со скаляром (shr)

Включим оптимизацию -O2

```
int image_is_dark_v3(uint8_t *img, int width, int height)
{
    int count = 0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            count += (img[i * width + j] >= 128);
        }
    }
    return count < width * height / 2;
}
```

- **Условное выражение заменено на арифметическое**
- Значение типа пикселя в диапазоне [0, 255], поэтому целочисленное деление на 128 дает 0 либо 1 (округление вниз)
- На уровне оптимизации -O2 компилятор gcc автоматически устраняет простые ветвления (сравнение со скаляром) – замена на арифметическое выражение

Сгенерированный gcc
код совпадает
с v2 (shr al, 7)

```
[0x000013ad]
mov eax, dword [i]
imul eax, dword [width]
mov edx, eax
mov eax, dword [j]
add eax, edx
movsxd rdx, eax
mov rax, qword [img]
add rax, rdx
movzx eax, byte [rax]
shr al, 7
movzx eax, al
add dword [count], eax
add dword [j], 1
```


Устранение ветвлений: сравнение со скаляром (setcc)

- Если при делении на скаляр частное больше единицы, то можно применять инструкции установки и копирования с условием (setcc)

```
int count = 0;
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        count += (img[i * width + j] >= 50);
    }
}
```

```
movzx    eax, byte [rax]
cmp      al, 0x31      # сравнение img[i][j] с 49
seta     al           # al = (img[i][j] > 49) ? 1 : 0
movzx    eax, al
```

```
[0x000013ad]
mov      eax, dword [i]
imul    eax, dword [width]
mov      edx, eax
mov      eax, dword [j]
add     eax, edx
movsxd  rdx, eax
mov      rax, qword [img]
add     rax, rdx
movzx   eax, byte [rax]
cmp     al, 0x31
seta    al
movzx   eax, al
add     dword [count], eax
add     dword [j], 1
```

- **seta** – set byte if above (CF=0 and ZF=0)
- Set byte if above {above, below, equal, greater, less, not above, not below, not less, not equal, zero, overflow, parity, parity odd, parity even, sign, not zero, sign, not sign, ...}

Устранение ветвлений: сравнение со скаляром (expr < scalar)

```
int image_is_light_v2(uint8_t *img, int width, int height)
{
    int count = 0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            count += (img[i * width + j] < 128);
        }
    }
    return count < width * height / 2;
}
```

- Ветвление заменено на арифметическое выражение

```
# img[i][j] >= 128
movzx  eax, byte [rax]
shr    al, 7
movzx  eax, al
add    dword [count], eax
```



```
# img[i][j] < 128
movzx  eax, byte [rax]
not    eax
shr    al, 7
movzx  eax, al
add    dword [count], eax
```

Пример: $img[i][j] = 129$
 $eax = 129 = 1000001_2$
 $shr\ al, 7 \Rightarrow al = 00000001_2$
 $eax = 1$

Пример: $img[i][j] = 100$
 $eax = 100 = 01100100_2$
 $not\ eax = 1111...10011011_2, al = 10011011_2$
 $shr\ al, 7 \Rightarrow al = 00000001_2 = 1$
 $eax = 1$

```
[0x0000132b]
mov     eax, dword [i]
imul   eax, dword [width]
mov     edx, eax
mov     eax, dword [j]
add     eax, edx
movsxd  rdx, eax
mov     rax, qword [img]
add     rax, rdx
movzx   eax, byte [rax]
not     eax
shr     al, 7
movzx   eax, al
add     dword [count], eax
add     dword [j], 1
```

Устранение ветвлений: сравнение со скаляром (clang/llvm)

```
int image_is_dark_v3(uint8_t *img, int width, int height)
{
    int count = 0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            count += (img[i * width + j] >= 128);
        }
    }
    return count < width * height / 2;
}
```

- **clang** -g -o prog ./prog.c
- Условное выражение заменено на инструкцию setge

```
movzx    eax, byte [rax + rcx]
cmp      eax, 0x80
setge    al
and      al, 1
movzx    eax, al
add      eax, dword [var_14h]
```

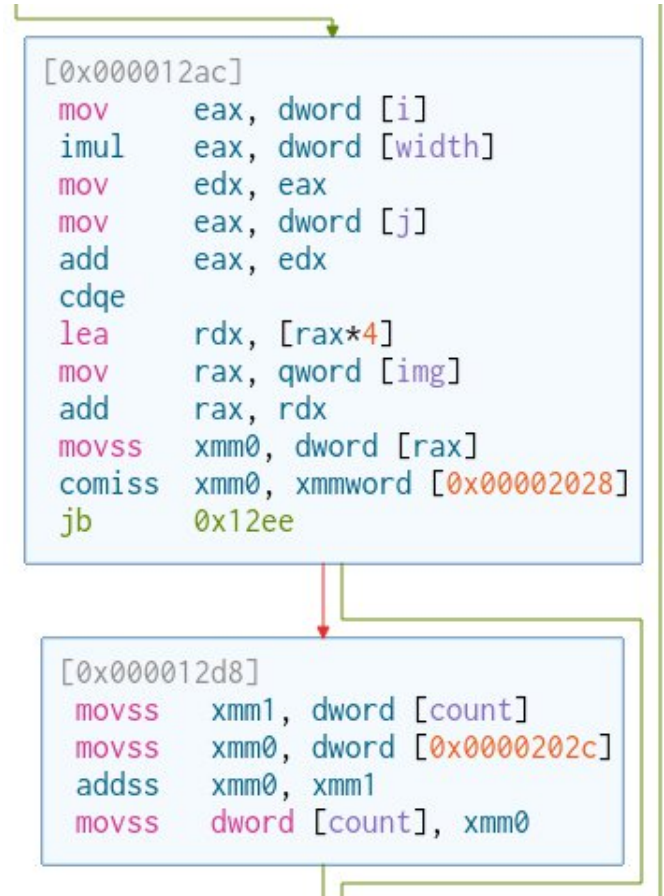
- В версии v2 (`img[i * width + j] >> 7`) Clang/LLVM заменяет условное выражение на операцию (`sar eax, 7`)

```
[0x0000135b]
mov     rax, qword [var_8h]
mov     ecx, dword [var_18h]
imul   ecx, dword [var_ch]
add     ecx, dword [var_1ch]
movsxd  rcx, ecx
movzx   eax, byte [rax + rcx]
cmp     eax, 0x80
setge   al
and     al, 1
movzx   eax, al
add     eax, dword [var_14h]
mov     dword [var_14h], eax
mov     eax, dword [var_1ch]
add     eax, 1
mov     dword [var_1ch], eax
jmp     0x134f
```

Устранение ветвлений: сравнение с вещественным скаляром IEEE 754 (float, double)

```
float image_is_dark(float *img, int width, int height)
{
    float count = 0.0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (img[i * width + j] >= 128.0) {
                count += 2;
            }
        }
    }
    return count < width * height / 2.0;
}
```

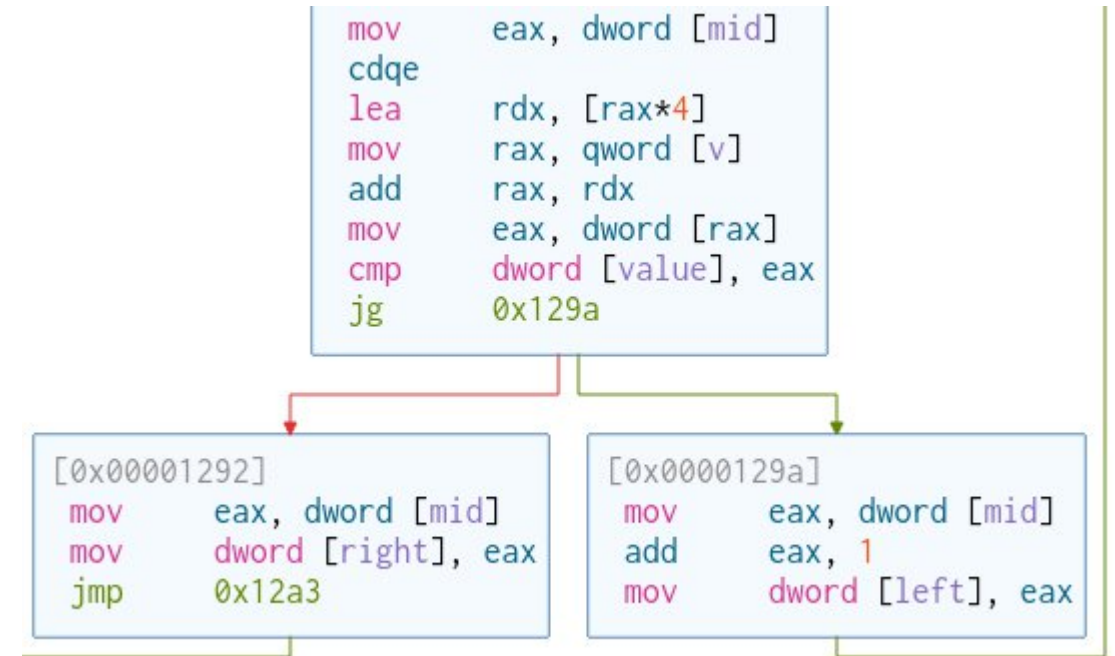
- Ветвление не устранено!
- comiss – сравнение векторного регистра xmm0 и константы 128.0 (константа (4 байта) хранится в секции .rodata)
- jnb – условный переход по результату сравнения
- Включение оптимизаций -O2 компилятора gcc не приводит к устранению условного выражения со скаляром типа float



float 128.0
binary: 0x43 = 67 = 01000011₂

Устранение ветвлений: бинарный поиск

```
/* lower_bound: Returns the first elem not less than value */
int lower_bound(int *v, int n, int value)
{
    int left = 0, right = n - 1;
    while (left < right) {
        int mid = (left + right) / 2;
        if (v[mid] >= value)
            right = mid;
        else
            left = mid + 1;
    }
    return v[left];
}
```



- `jg` – условный переход по результату сравнения

Устранение ветвлений: бинарный поиск (branchless)

```
int lower_bound_v2(int *v, int n, int value)
{
    int left = 0, right = n - 1;
    while (left < right) {
        int mid = (left + right) / 2;
        int ftrue = (v[mid] >= value);
        int ffalse = ftrue ^ 1;
        right = ftrue * mid + ffalse * right;
        left = ftrue * left + ffalse * (mid + 1);
    }
    return v[left];
}
```

- Замена условного выражения предикатом (predication)
- ftrue устанавливается с использованием инструкции setle

if (cond) then $a = \text{cond} * x +$ // Установить новое значение
 $a = x$ $(\text{cond} - 1) * a$ // Сохранить старое

```
mov     dword [mid], eax
mov     eax, dword [mid]
cdq
lea     rdx, [rax*4]
mov     rax, qword [v]
add     rax, rdx
mov     eax, dword [rax]
cmp     dword [value], eax
setle  al
movzx   eax, al
mov     dword [ftrue], eax
mov     eax, dword [ftrue]
xor     eax, 1
mov     dword [ffalse], eax
mov     eax, dword [ftrue]
imul   eax, dword [mid]
mov     edx, eax
mov     eax, dword [ffalse]
imul   eax, dword [right]
add     eax, edx
mov     dword [right], eax
mov     eax, dword [ftrue]
imul   eax, dword [left]
mov     edx, eax
mov     eax, dword [mid]
add     eax, 1
imul   eax, dword [ffalse]
add     eax, edx
mov     dword [left], eax
```

Устранение ветвлений: бинарный поиск (gcc -O2)

```
int lower_bound_v2(int *v, int n, int value)
{
    int left = 0, right = n - 1;
    while (left < right) {
        int mid = (left + right) / 2;
        int ftrue = (v[mid] >= value);
        int ffalse = ftrue ^ 1;
        right = ftrue * mid + ffalse * right;
        left = ftrue * left + ffalse * (mid + 1);
    }
    return v[left];
}
```

- **Условное выражение заменено на арифметические**
 - ftrue устанавливается с использованием инструкции setge
 - ffalse устанавливается инструкцией setl
- **cmov** – условное копирование (conditional move)
 - cmovge dest, src – conditional move if greater or equal
 - cmovl dest, src – conditional move if less

```
[0x000012e8]
lea ecx, [rdx + rsi]
mov eax, ecx
shr eax, 0x1f
add eax, ecx
sar eax, 1
movsxd rcx, eax
mov ecx, dword [rdi + rcx*4] ; arg1
cmp ecx, r9d
setge r8b # ftrue
cmovge esi, r10d
movzx r8d, r8b
imul r8d, eax
add esi, r8d
cmp ecx, r9d
cmovl edx, r10d
add eax, 1
cmp ecx, r9d
setl cl # ffalse
movzx ecx, cl
imul eax, ecx
add edx, eax
cmp esi, edx
jg 0x12e8
```

Устранение условного выражения цикла for

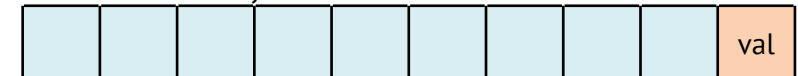
```
// find: Returns 1 if val is present in array vec
int find(int *vec, size_t size, int val)
{
    for (size_t i = 0; i < size; i++) {           // cmp + je
        if (val == vec[i])                       // cmp + jne
            return 1;
    }
    return 0;
}
```

- Функция find выполняет линейный поиск элемента в массиве
- $2 * \text{size}$ условных переходов (je, jne)

Устранение условного выражения цикла for

```
// find: Returns 1 if val is present in array vec
int find_sentinel(int *vec, size_t size, int val)
{
    // Real size of vec is size + 1
    vec[size] = val;
    for (size_t i = 0; ; i++) {
        if (val == vec[i]) { // cmp + jne
            if (i == size - 1) { // cmp + setne
                return 0;
            } else {
                return 1;
            }
        }
    }
}
```

vec: size = 9, allocated size = 10



- Если имеется возможность выделить в конце массива vec дополнительный элемент, то его содержимое можно использовать как *сигнальный элемент* о достижении конца массива (элемент "страж", sentinel)
- Цикл for заменяется на бесконечный, устраняется необходимость условного перехода (je)
- size условных переходов (jne)

Таблицы поиска (lookup table)

```
enum {  
    BLOCK_T1=0, BLOCK_T2=1, BLOCK_T3=2, BLOCK_T4=3, BLOCK_T5=4, BLOCK_COUNT=5  
}
```

```
int get_block_type(int block)  
{  
    if (block >= 0 && block <= 64) return BLOCK_T1;  
    if (block > 64 && block <= 128) return BLOCK_T2;  
    if (block > 128 && block <= 150) return BLOCK_T3;  
    if (block > 150 && block < 190) return BLOCK_T4;  
    if (block >= 190 && block < 256) return BLOCK_T5;  
    return -1;  
}
```

```
void blocks(int *blocks, int n, int *freq)  
{  
    int i = 0;  
    while (i < n) {  
        int type = get_block_type(blocks[i++]);  
        freq[type]++;  
    }  
}
```

```
$ perf record -e branch-misses ./blocks
```

```
Samples: 9K of event 'branch-misses', Event count (approx.): 95226104  
Overhead Command Shared Object Symbol  
75,93% blocks blocks [.] blocks  
20,59% blocks blocks [.] get_block_type  
3,36% blocks libc.so.6 [.] __vfscanf_internal  
0,04% blocks blocks [.] main  
0,02% blocks [unknown] [k] 0xffffffff93769655  
0,01% blocks [unknown] [k] 0xffffffff937052c3  
0,00% blocks [unknown] [k] 0xffffffff937206ee
```

```
8,18      cpl    $0xbd,-0x4(%rbp)  
↓ jg     66  
16,93     mov    $0x3,%eax  
0,12      ↓ jmp   84  
          if (block >= 190 && block < 256) return BLOCK_T5;  
21,69     66:  cpl    $0xbd,-0x4(%rbp)  
          ↓ jle   7f  
6,32     cpl    $0xff,-0x4(%rbp)  
          ↓ jg     7f
```

Замена ветвлений таблицей поиска (lookup table)

```
int block_types[256] = {
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    3, 3, 3, 3, 3,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
};

int get_block_type(int block)
{
    if (block < (sizeof(block_types) / sizeof(block_types[0])))
        return block_types[block];
    return -1;
}

void blocks(int *blocks, int n, int *freq)
{
    int i = 0;
    while (i < n) {
        int type = get_block_type(blocks[i++]);
        freq[type]++;
    }
}
```

- Boost: boost::icl::interval_map
- LLVM: IntervalMap

Оптимизация инвариантных ветвлений

```
for (i = 0; i < 10; i++) {  
    if (value > 10)  
        data++;  
    else  
        data--;  
}
```

- **Инвариантное ветвление** – ветвление, направления которого не зависят от индуктивных переменных цикла (от счетчика цикла)

Сколько будет выполнено условных переходов?

Оптимизация инвариантных ветвлений

```
for (i = 0; i < 10; i++) {  
    if (value > 10)  
        data++;  
    else  
        data--;  
}
```

1. 0 < 10
2. value > 10
3. 1 < 10
4. value > 10
5. 2 < 10
6. value > 10
7. 3 < 10
8. value > 10
9. 4 < 10
10. value > 10
11. 5 < 10
12. value > 10

13. 6 < 10
14. value > 10
15. 7 < 10
16. value > 10
17. 8 < 10
18. value > 10
19. 9 < 10
20. value > 10
21. 10 < 10

21 условный переход

Вынос инвариантных ветвлений из цикла

```
for (i = 0; i < 10; i++) {  
    if (value > 10)  
        data++;  
    else  
        data--;  
}
```



```
if (value > 10) {  
    for (i = 0; i < 10; i++)  
        data++;  
} else {  
    for (i = 0; i < 10; i++)  
        data--;  
}
```

20 условных переходов

(условие цикла и ветвление в его теле)

value > 10 – инвариантное условие
(не зависит от параметра цикла i)

12 условных переходов

Меньше обращений к модулю предсказания переходов

- value > 10
- 0 < 10
- 1 < 10
- ...
- 10 < 10

Вынос инвариантных ветвлений из цикла

```
void blend(int size, int blend, float *src, float *dest, float *src_1)
{
    for (int j = 0; j < size; j++) {
        if (blend == 255)
            dest[j] = src_1[j];
        else if (blend == 0)
            dest[j] = src_2[j];
        else
            dest[j] = (src_1[j] * blend + src_2[j] * (255 - blend)) / 256;
    }
}
```

Вынос инвариантных ветвлений из цикла

```
void blend(int size, int blend, float *src, float *dest, float *src_1)
{
    if (blend == 255)
        for (int j = 0; j < size; j++)
            dest[j] = src_1[j];

    else if (blend == 0)
        for (int j = 0; j < size; j++)
            dest[j] = src_2[j];

    else
        for (int j = 0; j < size; j++)
            dest[j] = (src_1[j] * blend + src_2[j] * (255 - blend)) / 256;
}
```

- Инвариантное ветвление вынесли за цикл
- Сократили число переходов + возможность векторизации кода

Раскрутка цикла (loop unrolling)

- **Раскрутка цикла** на k итераций – тиражирование тела цикла k раз
- **Плюсы:**
 - Сокращается количество условных переходов (вычисления условного выражения)
 - Позволяет обеспечить параллельное выполнение инструкций нового тела цикла на суперскалярном ядре (если инструкции не зависимы по данным)
- **Минусы:** увеличивает размер кода и количество используемых регистров процессора

```
// Исходный цикл
for (int i = 0; i < n; i++) {
    z[i] = a * x[i] + y[i]
}
```



```
// Раскрученный цикл из n / k итераций
int i;
for (i = 0; i + k - 1 < n; i += k) {
    z[i] = a * x[i] + y[i]
    z[i + 1] = a * x[i + 1] + y[i + 1]
    z[i + 2] = a * x[i + 2] + y[i + 2]
    ...
    z[i + k - 1] = a * x[i + k - 1] + y[i + k - 1]
}

// Остаток итераций n % k
for (; i < n; i++) {
    z[i] = a * x[i] + y[i]
}
```

Раскрутка цикла (loop unrolling)

```
int vec_sum(int *vec, int n)
{
    int s = 0;
    for (int i = 0; i < n; i++) {
        s += vec[i];
    }
    return s;
}
```



```
int vec_sum_v2(int *vec, int n)
{
    unsigned int i;
    int s = 0;

    for (i = 0; i + 3 < n; i += 4) {
        s = s + vec[i];
        s = s + vec[i + 1];
        s = s + vec[i + 2];
        s = s + vec[i + 3];
    }

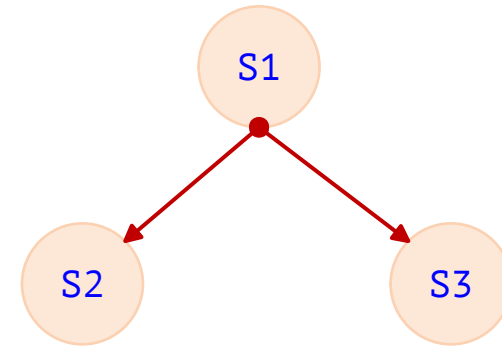
    for (; i < n; i++) {
        s += vec[i];
    }
    return s;
}
```

Конфликты данных (Data Hazards)

- Текущий шаг конвейера не может быть выполнен, так как зависит от результатов выполнения предыдущего шага
- Возможные причины:
 - **Read after Write (RAW)** – True dependency
i1: $R2 = R1 + R3$
i2: $R4 = R2 + R3$
 - **Write after Read (WAR)** – Anti-dependency
 $R4 = R1 + R3$
 $R3 = R1 + R2$
 - **Write after Write (WAW)** – Output dependency
 $R2 = R4 + R7$
 $R2 = R1 + R3$

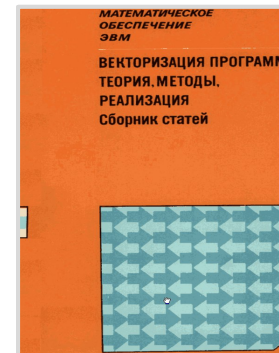
Конфликты данных (Data Hazards)

S1: $A = B + C$
S2: $D = A + 2$
S3: $E = A + 3$



Граф зависимостей по данным
(data-dependence graph)

- S2 зависит от S1 – S1 и S2 нельзя выполнять параллельно
- S3 зависит от S1 – S1 и S3 нельзя выполнять параллельно
- S2 и S3 можно выполнять параллельно



Векторизация программ. Теория, методы, реализация
(сборник статей). – М.: Мир, 1991. – 275 с.

Раскрутка цикла (loop unrolling)

```
int vec_sum_v2(int *vec, int n)
{
    unsigned int i;
    int s = 0;

    for (i = 0; i + 3 < n; i += 4) {
        s = s + vec[i];
        s = s + vec[i + 1];
        s = s + vec[i + 2];
        s = s + vec[i + 3];
    }

    for (; i < n; i++) {
        s += vec[i];
    }
    return s;
}
```



```
int vec_sum_v3(int *vec, int n)
{
    unsigned int i;
    int s = 0, t1 = 0, t2 = 0, t3 = 0;

    for (i = 0; i + 3 < n; i += 4) {
        s = s + vec[i];
        t1 = t1 + vec[i + 1];
        t2 = t2 + vec[i + 2];
        t3 = t3 + vec[i + 3];
    }
    t1 += t2 + t3;
    for (; i < n; i++) {
        s += vec[i];
    }
    return s + t1;
}
```

- **Зависимость по данным** (переменная s) препятствует параллельному выполнению сложения на независимых АЛУ

- **Зависимость по данным устранена** суммированием во временные переменные t1, t2, t3

Устранение ветвлений с использованием SIMD-инструкций

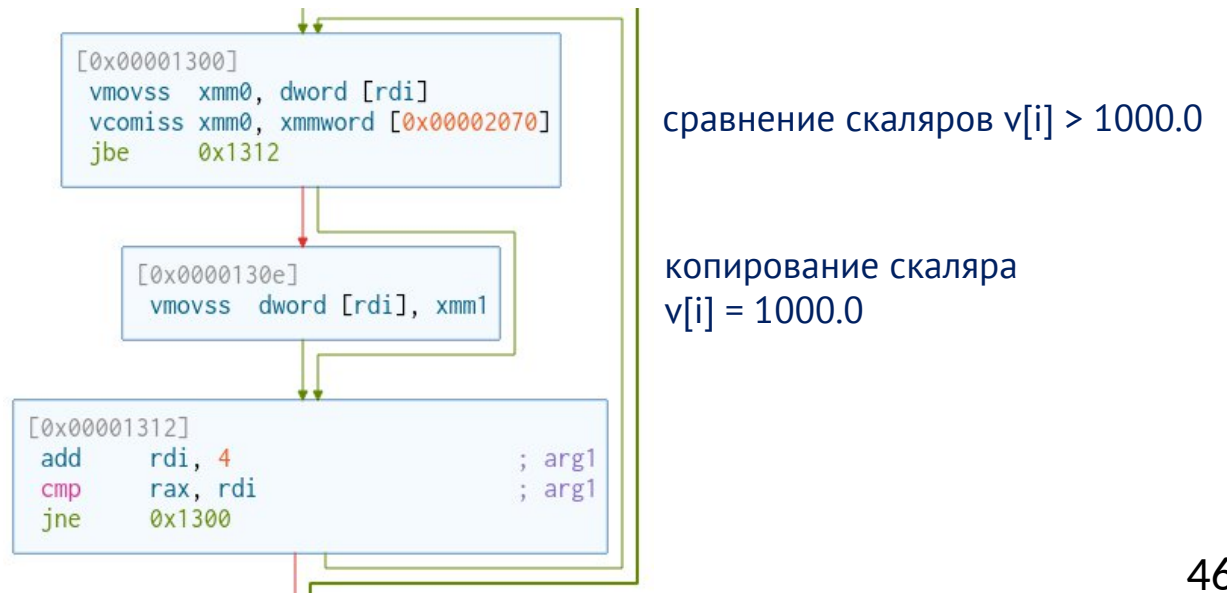
```
enum { N = 10000007 };

void vtrunc(float *v, int n)
{
    for (int i = 0; i < n; i++) {
        if (v[i] > 1000.0) {
            v[i] = 1000.0;
        }
    }
}

double run()
{
    float *v = malloc(sizeof(*v) * N);
    vtrunc(v, N);

    // ...
}
```

- Элементы массива имеют вещественный тип одинарной точности (float)
- Компилятор (gcc 11.2) генерирует код с ветвлением в цикле (сравнение скаляров vcomiss и переход jbe)
- gcc -O2 -mavx -o prog ./prog.c



Устранение ветвлений с использованием SIMD-инструкций: AVX, float

1. Формируем вектор из 8 скаляров 1000.0

```
__m256 v1000 = _mm256_set1_ps(1000.0);
```

	mm256_set1_ps							
v1000	1000	1000	1000	1000	1000	1000	1000	1000

2. Загружаем в векторный регистр 8 элементов v[i:i+7]

```
__m256 val = _mm256_load_ps(&v[i]);
```

	mm256_load_ps							
val	940	2002	1100	23	3000	1900	1002	1000

3. Выполняем векторное сравнение: v[i:i+7] > [1000.0, 1000.0, ..., 1000.0]

```
__m256 mask = _mm256_cmp_ps(val, v1000, _CMP_GT_0Q)
```

результат сравнения – вектор mask[0:7], в котором mask[i] = v[i] > 0 ? 0xFFFFFFFF : 0

	mm256_cmp_ps							
mask	0	FF..FF	FF..FF	23	FF..FF	FF..FF	FF..FF	0

4. Формируем результат из элементов, для которых выполнено условие v[i] > 0, накладываем маску на вектор v1000 (AND)

```
__m256 true_vec = _mm256_and_ps(mask, v1000)
```

	_mm256_and_ps							
true_vec	0	1000	1000	0	1000	1000	1000	0

- true_vec[0] = 0x0 AND 1000 = 0
- true_vec[1] = 0xFF..FF AND 1000 = 1000

5. Формируем результат из элементов, для которых не выполнено условие v[i] <= 0, накладывем маску на вектор val (AND NOT)

```
__m256 false_vec = _mm256_andnot_ps(mask, val)
```

	_mm256_andnot_ps							
false_vec	940	0	0	23	0	0	0	1000

- true_vec[0] = NOT(0x0) AND 940 = 1 AND 940 = 940
- true_vec[1] = NOT(0xFF..FF) AND 2002 = 0

6. Объединяем результаты и записываем в вектор v[i:i+7]

```
_mm256_store_ps(&v[i],  
              _mm256_or_ps(true_vec, false_vec))
```

	_mm256_or_ps							
v[i:i+7]	940	1000	1000	23	1000	1000	1000	1000

- v[i] = 0 OR 940 = 940
- v[i+1] = 1000 OR 0 = 1000

Устранение ветвлений с использованием SIMD-инструкций: AVX, float

```
void vtrunc_avx(float *v, int n)
{
    __m256 *vec = (__m256 *)v;
    __m256 v1000 = _mm256_set1_ps(1000);
    int k = n / 8;

    for (int i = 0; i < k; i++) {
        __m256 val = _mm256_load_ps((float *)&vec[i]);
        __m256 mask = _mm256_cmp_ps(val, v1000, _CMP_GT_00);
        __m256 true_vec = _mm256_and_ps(mask, v1000);
        __m256 false_vec = _mm256_andnot_ps(mask, val);
        vec[i] = _mm256_or_ps(true_vec, false_vec);
    }

    for (int i = k * 8; i < n; i++) {
        if (v[i] > 1000) {
            v[i] = 1000;
        }
    }
}
```

- Адрес `v` должен быть выравнен на границу кратную размеру векторного регистра – 32 байта
- Выделение памяти с заданным выравниванием
 - `float v[N] __attribute__((aligned(32)))`;
 - `#include <malloc.h>`
 - `void *_mm_malloc(size_t size, size_t align)`
 - `void _mm_free(void *p)`

 - `#include <stdlib.h>`
 - `int posix_memalign(void **memptr, size_t alignment, size_t size)`;

 - `#include <stdlib.h>`
 - `void *aligned_alloc(size_t alignment, size_t size)`;

Устранение ветвлений с использованием SIMD-инструкций: loop peeling

- Peeling loop – скалярно вычисляет часть элементов для достижения корректного выравнивания

```
void vtrunc_avx_peeled(float *v, int n) {
    int simd_width_bytes = sizeof(__m256); // Размер векторного регистра в байтах (32B)
    int simd_width_f32 = simd_width_bytes / sizeof(*v); // Количество элементов типа float в векторном регистре (8)
    int misalign_bytes = (uintptr_t)v & (simd_width_bytes - 1); // Количество байт за границей выравнивания (v % 32)

    // Обрабатываем первые элементы скалярно, пока не достигнем элемента с корректно выровненным адресом
    int peeled_iters = 0;
    if (misalign_bytes > 0) {
        peeled_iters = (simd_width_bytes - misalign_bytes) / sizeof(*v);
        for (int i = 0; i < peeled_iters; i++) {
            if (v[i] > 1000) v[i] = 1000;
        }
    }

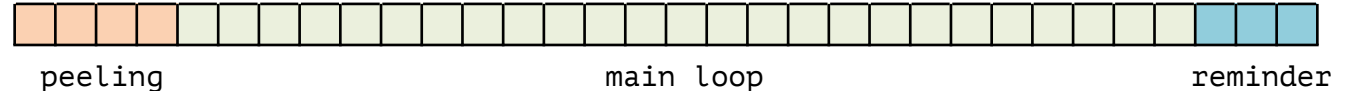
    int main_iters = n - ((n - peeled_iters) & (simd_width_f32 - 1));
    __m256 v1000 = _mm256_set1_ps(1000);
    for (int i = peeled_iters; i < main_iters; i += simd_width_f32) {
        __m256 val = _mm256_load_ps(&v[i]);
        __m256 mask = _mm256_cmp_ps(val, v1000, _CMP_GT_OQ);
        __m256 true_vec = _mm256_and_ps(mask, v1000);
        __m256 false_vec = _mm256_andnot_ps(mask, val);
        _mm256_store_ps(&v[i], _mm256_or_ps(true_vec, false_vec));
    }

    for (int i = main_iters; i < n; i++) {
        if (v[i] > 1000) v[i] = 1000;
    }
}
```

- # AVX, float (8 байт)
- n = 1000007
- v = malloc(sizeof(*v) * n);
- # malloc result: v % 32 == 16

- misalign_bytes = 16
- peeled_iters = 4
- main_iters = 1000004

v = 0x7f043a3da010



Устранение ветвлений с использованием SIMD-инструкций: AVX, float + blendv

```
void vtrunc_avx_peeled(float *v, int n)
{
    // Alignment loop
    if (misalign_bytes > 0) {
        ...
    }

    // Main loop
    int main_iters = n - ((n - peeled_iters) & (simd_width_f32 - 1));
    __m256 v1000 = _mm256_set1_ps(1000);
    for (int i = peeled_iters; i < main_iters; i += simd_width_f32) {
        __m256 val = _mm256_load_ps((__mm256 *)&v[i]);
        __m256 mask = _mm256_cmp_ps(val, v1000, _CMP_GT_OQ);
        _mm256_store_ps((__mm256 *)&v[i],
            _mm256_blendv_ps(val, v1000, mask));
        // Слияние по маске
    }

    // Reminder loop
    for (int i = main_iters; i < n; i++) {
        ...
    }
}
```

[*] <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/>

`_mm256_cmp_ps()`

Architecture	Latency	Throughput (CPI)
Alderlake	4	0.5
Icelake Intel Core	4	0.5
Icelake Xeon	4	0.5
Skylake	4	0.5

`_mm256_blendv_ps()`

Architecture	Latency	Throughput (CPI)
Alderlake	3	1
Icelake Intel Core	-	1
Icelake Xeon	2	1
Skylake	2	0.66

`_mm256_{and, andnot, or}_ps()`

Architecture	Latency	Throughput (CPI)
Alderlake	1	0.33
Icelake Intel Core	1	0.33
Icelake Xeon	1	0.33
Skylake	1	0.33

Устранение ветвлений с использованием SIMD-инструкций: AVX, float + min

```
void vtrunc_avx_peeked(float *v, int n)
{
    if (misalign_bytes > 0) {
        // Peeling loop
    }

    // Main loop
    int main_iters = n - ((n - peeled_iters) & (simd_width_f32 - 1));
    __m256 v1000 = _mm256_set1_ps(1000);

    for (int i = peeled_iters; i < main_iters; i += simd_width_f32) {
        __m256 val = _mm256_load_ps(&v[i]);
        _mm256_store_ps(&v[i], _mm256_min_ps(val, v1000));
    }

    // Reminder loop
    for (int i = main_iters; i < n; i++) {
        if (v[i] > 1000) {
            v[i] = 1000;
        }
    }
}
```

_mm256_min_ps()

Architecture	Latency	Throughput (CPI)
Alderlake	4	0.5
Icelake Intel Core	4	0.5
Icelake Xeon	4	0.5
Skylake	4	0.5

Устранение ветвлений с использованием SIMD-инструкций: AVX, int

```
void vtrunc_avx_peeled(int *v, int n)
{
    int simd_width_bytes = sizeof(__m256i);           // 32
    int simd_width_i32 = simd_width_bytes / sizeof(*v); // 8
    int misalign_bytes = (uintptr_t)v & (simd_width_bytes - 1); // v mod 32
    int peeled_iters = 0;
    // Peeling loop
    if (misalign_bytes > 0) {
        peeled_iters = (simd_width_bytes - misalign_bytes) / sizeof(*v);
        for (int i = 0; i < peeled_iters; i++) {
            if (v[i] > 1000)
                v[i] = 1000;
        }
    }

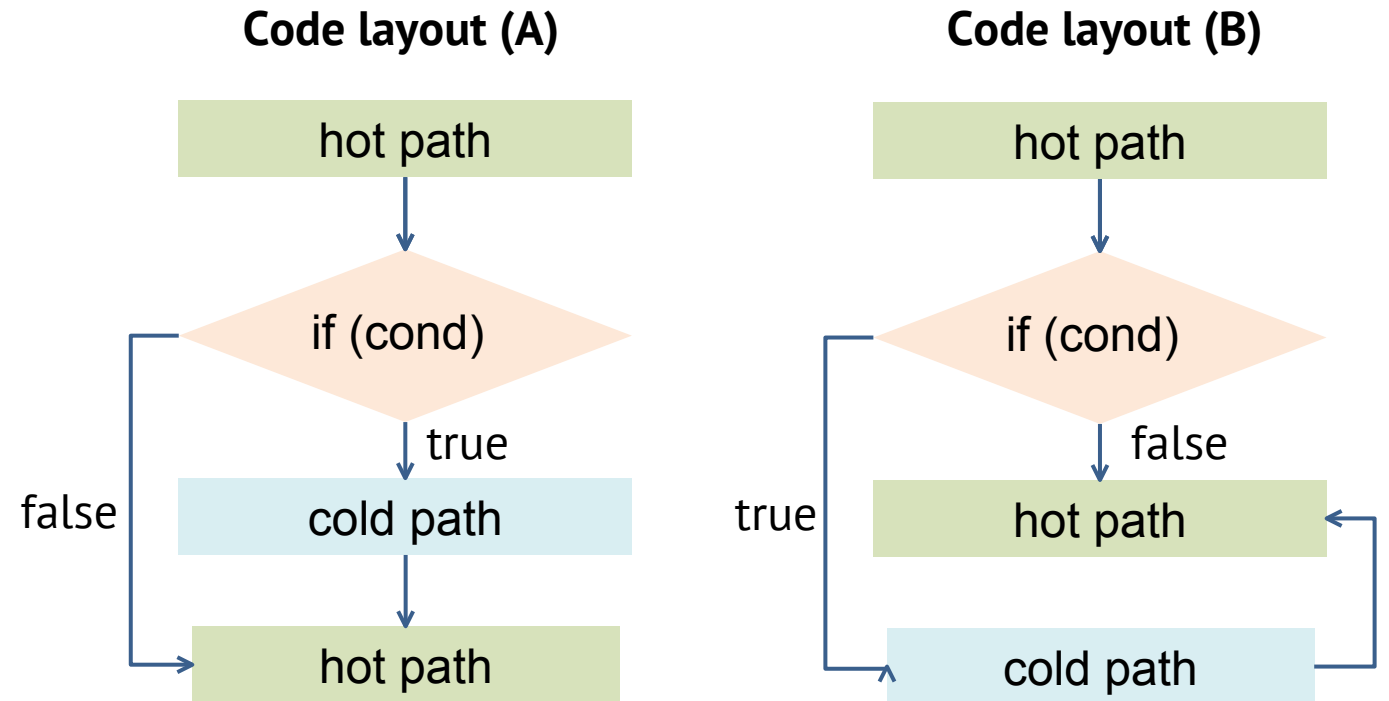
    // Main loop
    int main_iters = n - ((n - peeled_iters) & (simd_width_i32 - 1));
    __m256i v1000 = _mm256_set1_epi32(1000);           // Broadcast 32-bit int 1000 to all 32 elements of v1000
    for (int i = peeled_iters; i < main_iters; i += simd_width_i32) {
        __m256i val = _mm256_load_si256((__m256i *)&v[i]);
        __m256i mask = _mm256_cmpgt_epi32(val, v1000); // Compare val > v1000
        __m256i true_vec = _mm256_and_si256(mask, v1000);
        __m256i false_vec = _mm256_andnot_si256(mask, val);
        _mm256_store_si256((__m256i *)&v[i], _mm256_or_si256(true_vec, false_vec));
    }
    for (int i = main_iters; i < n; i++) {
        if (v[i] > 1000)
            v[i] = 1000;
    }
}
```

Размещение блоков кода

- "Горячий путь" (hot code path) – путь в графе управления, на котором программа проводит больше всего времени (содержит вычислительно сложные базовые блоки, часто выполняется)

```
// hot code path
if (cond) {
    // cold code path
}
// hot code path
```

- Какой вариант предпочтительнее – А или В?
- Если cond вероятнее всего true, то эффективнее А – меньше переходов, предвыборка в L1i
- Если cond вероятнее всего false, то эффективнее В



Аннотирование вероятности ветвлений (likely/unlikely)

- GCC `__builtin_expect()`

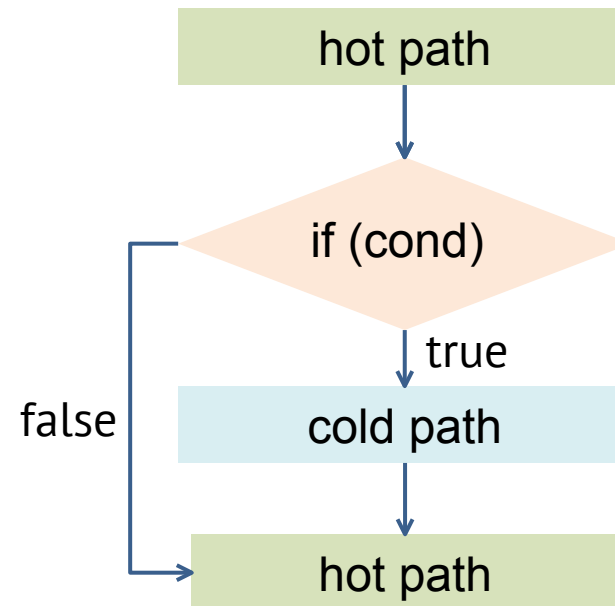
```
#define likely(expr) __builtin_expect((expr), 1)  
#define unlikely(expr) __builtin_expect((expr), 0)
```

```
if (likely(p != NULL)) {  
    // process p  
}
```

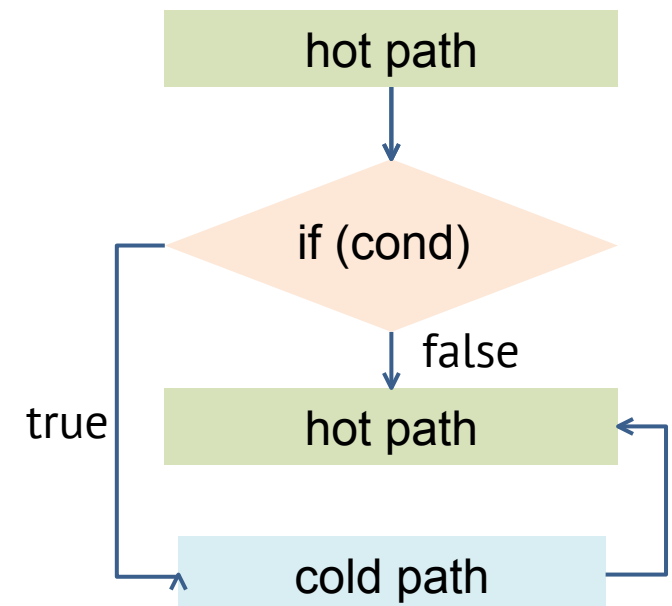
- C++20

```
if (p != NULL) [[likely]] {  
    // process p  
}
```

Code layout A



Code layout B



Аннотирование ветвлений (likely/unlikely)

```
// V1
int f(int n, int k)
{
    int s = 0;
    for (int i = 0; i < n; i++) {
        if ((k % 2) == 0)
            s += pop(i);
        else
            s -= pop(i);
    }
    return s;
}

// V2 с аннотированием
int f(int n, int k)
{
    int s = 0;
    for (int i = 0; i < n; i++) {
        if (unlikely((k % 2) == 0))
            s += pop(i);
        else
            s -= pop(i);
    }
    return s;
}
```

```
# V1 – без аннотирования условий
f:
.LFB41:
    testl    %edi, %edi
    jle     .L12
    andl    $1, %esi
    xorl    %ecx, %ecx
    xorl    %r8d, %r8d

.L11:
    movl    %ecx, %edx
    movl    %ecx, %eax
    shrl    %edx
    andl    $1431655765, %edx
    ...
    subl    %eax, %r8d
    testl   %esi, %esi
    cmovne %edx, %r8d
    addl    $1, %ecx
    cmpl    %ecx, %edi
    jne     .L11
    movl    %r8d, %eax
    ret

.L12:
    xorl    %r8d, %r8d
    movl    %r8d, %eax
    ret
```

```
# V2 – с аннотацией unlikely
f:
.LFB41:
    testl    %edi, %edi
    jle     .L12
    notl    %esi
    xorl    %ecx, %ecx
    xorl    %r8d, %r8d
    andl    $1, %esi

.L11:
    movl    %ecx, %edx
    movl    %ecx, %eax
    shrl    %edx
    andl    $1431655765, %edx
    ...
    subl    %eax, %r8d
    testl   %esi, %esi
    cmovne %edx, %r8d
    addl    $1, %ecx
    cmpl    %ecx, %edi
    jne     .L11
    movl    %r8d, %eax
    ret

.L12:
    xorl    %r8d, %r8d
    movl    %r8d, %eax
    ret
```

Дизассемблирование с помощью dgb

```
$ gdb -batch -ex "file ./prog" -ex "disassemble f"
```

```
Dump of assembler code for function f:
```

```
0x00000000000001274 <+4>:      test   %edi,%edi
0x00000000000001276 <+6>:      jle   0x12e0 <f+112>
0x00000000000001278 <+8>:      not   %esi
0x0000000000000127a <+10>:     xor   %ecx,%ecx
0x0000000000000127c <+12>:     xor   %r8d,%r8d
0x0000000000000127f <+15>:     and   $0x1,%esi

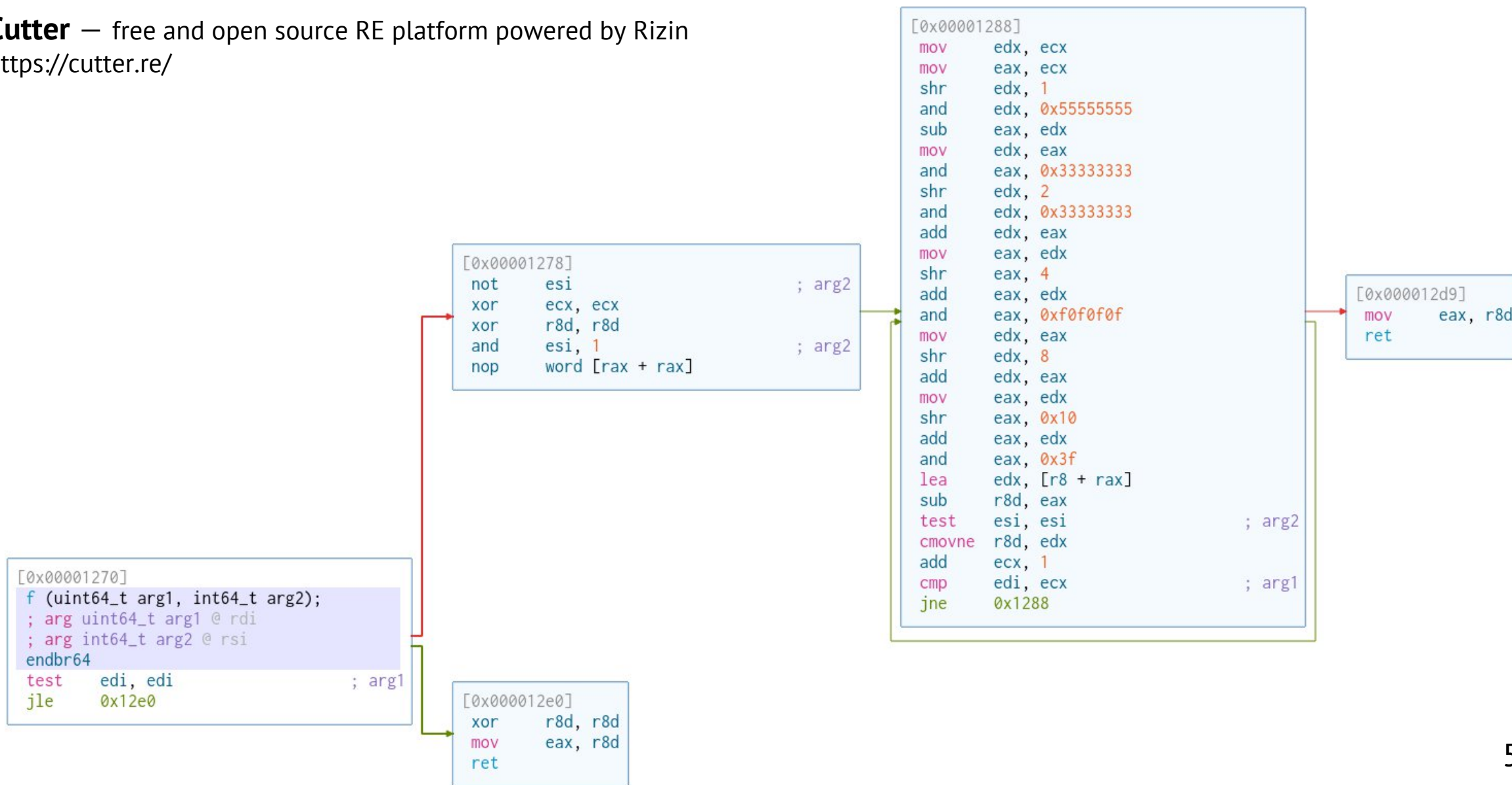
0x00000000000001282 <+18>:     nopw  0x0(%rax,%rax,1)
0x00000000000001288 <+24>:     mov   %ecx,%edx
0x0000000000000128a <+26>:     mov   %ecx,%eax
0x0000000000000128c <+28>:     shr   %edx
0x0000000000000128e <+30>:     and   $0x55555555,%edx
...
0x000000000000012c9 <+89>:     sub   %eax,%r8d
0x000000000000012cc <+92>:     test  %esi,%esi
0x000000000000012ce <+94>:     cmovne %edx,%r8d
0x000000000000012d2 <+98>:     add   $0x1,%ecx
0x000000000000012d5 <+101>:    cmp   %ecx,%edi
0x000000000000012d7 <+103>:    jne   0x1288 <f+24>
...
0x000000000000012e3 <+115>:    mov   %r8d,%eax
0x000000000000012e6 <+118>:    ret
```

```
End of assembler dump.
```


Дизассемблирование с помощью cutter + rizin

Cutter — free and open source RE platform powered by Rizin

<https://cutter.re/>



Профилирование программы с помощью gcc/gcov

1) Компиляция программы для сбора статистики выполнения ветвлений

```
$ gcc -g -fprofile-generate -fprofile-arcs -ftest-coverage -O2 -o prog ./prog.c
```

2) Запуск программы и сборка профиля

```
$ ./prog
```

```
# => prog.gcda, prog.gcov
```

3) Обработка профиля и генерация отчета

```
$ gcov -b ./prog.c
```

```
File 'prog.c'
```

```
Lines executed:92.00% of 25
```

```
Branches executed:100.00% of 5
```

```
Taken at least once:60.00% of 5
```

```
Calls executed:100.00% of 4
```

```
Creating 'prog.c.gcov'
```

4) Просмотр аннотированного исходного кода

```
$ cat prog.c.gcov
```

```
...
```

```
function f called 2 returned 100% blocks executed 75%
```

```
2: 23:int f(int n, int k)
```

```
-: 24:{
```

```
2: 25: int s = 0;
```

```
200000002: 26: for (int i = 0; i < n; i++) {
```

```
branch 0 taken 100%
```

```
branch 1 taken 1% (fallthrough)
```

```
200000000: 27: if (k == 0 || k == 2 || k == 4 || k == 8)
```

```
branch 0 taken 0%
```

```
branch 1 taken 100%
```

```
branch 2 taken 0%
```

```
#####: 28: s += pop(i);
```

```
-: 29: else if (k == 1 || k == 3 || k == 5 || k == 7)
```

```
200000000: 30: s -= pop(i);
```

```
-: 31: else
```

```
#####: 32: s += 1;
```

```
-: 33: }
```

```
2: 34: return s;
```

```
-: 35: }
```

Профилирование программ с помощью clang/llvm-cov

Компиляция программы для сбора статистики выполнения

```
$ clang -fprofile-instr-generate -fcoverage-mapping -O2 -o prog ./prog.c
```

```
$ ./prog # Запуск программы на типовых входных данных
```

Объединение отчетов профилирования

```
$ llvm-profdata merge -sparse *.profraw -o profdata.prof
```

```
$ llvm-cov show --show-branches=count --show-expansions \  
./prog -instr-profile=profdata.prof
```

Оптимизация по результатам профилирования в Clang/LLVM (profile-guided optimization)

```
23|      |int f(int n, int k)
24|      1|{
25|      1|      int s = 0;
26| 100M|      for (int i = 0; i < n; i++) {
-----
| Branch (26:21): [True: 100M, False: 1]
-----
27| 100M|          if (k == 0 || k == 2 || k == 4 || k == 8)
-----
| Branch (27:13): [True: 0, False: 100M]
| Branch (27:23): [True: 0, False: 100M]
| Branch (27:33): [True: 0, False: 100M]
| Branch (27:43): [True: 0, False: 100M]
-----
28|      0|              s += pop(i);
29| 100M|          else if (k == 1 || k == 3 || k == 5 || k == 7)
-----
| Branch (29:18): [True: 0, False: 100M]
| Branch (29:28): [True: 0, False: 100M]
| Branch (29:38): [True: 0, False: 100M]
| Branch (29:48): [True: 100M, False: 0]
-----
30| 100M|              s -= pop(i);
31|      0|          else
32|      0|              s += 1;
33| 100M|      }
34|      1|      return s;
35|      1|}
```

Оптимизация по результатам профилирования в GCC (profile-guided optimization)

Компиляция программы для сбора статистики выполнения

```
$ gcc -g -fprofile-generate -fprofile-arcs -ftest-coverage \  
-O2 -o prog ./prog.c
```

```
$ ./prog # Запуск программы на типовых входных данных
```

Компиляция и оптимизация программы с использованием собранной

статистики

```
$ gcc -fprofile-use -O2 -o prog ./prog.c
```

Оптимизация по результатам профилирования в Clang/LLVM (profile-guided optimization)

```
# Компиляция программы для сбора статистики выполнения  
$ clang -fprofile-instr-generate -O2 -o prog ./prog.c
```

```
$ ./prog      # Запуск программы на типовых входных данных
```

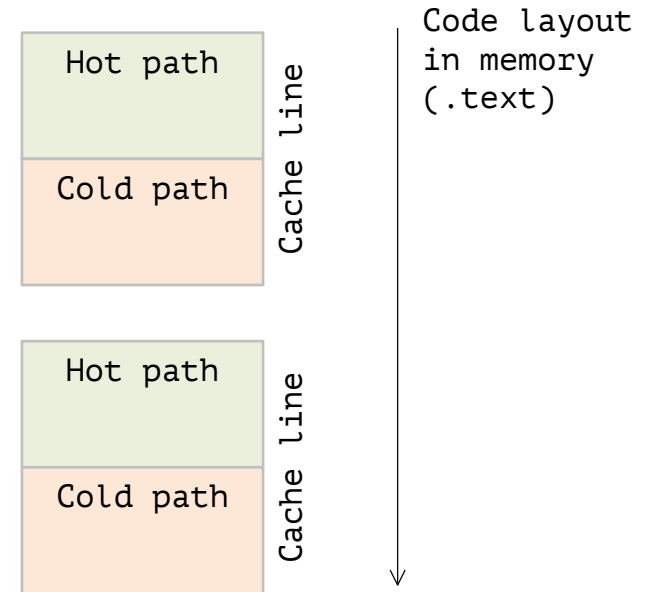
```
# Объединение отчетов профилирования  
$ llvm-profdata merge -output=profdata.prof *.profraw
```

```
# Компиляция и оптимизация программы с использованием собранной  
# статистики  
$ clang -fprofile-instr-use=profdata.prof -O2 -o prog ./prog.c
```

Разбиение больших функций (function splitting)

```
void process(void *data)
{
    // Hot code block
    if (error) {
        // Large cold code block (restoring, saving data)
    }

    // Hot code block
    if (error) {
        // Large cold code (restoring, saving data)
    }
}
```



- В больших функциях базовые блоки "горячего" и "холодного" путей могут попадать в одну и ту же строку кеш-памяти инструкций, что приводит к увеличению числа промахов
- Целесообразно плотнее размещать базовые блоки на горячем пути

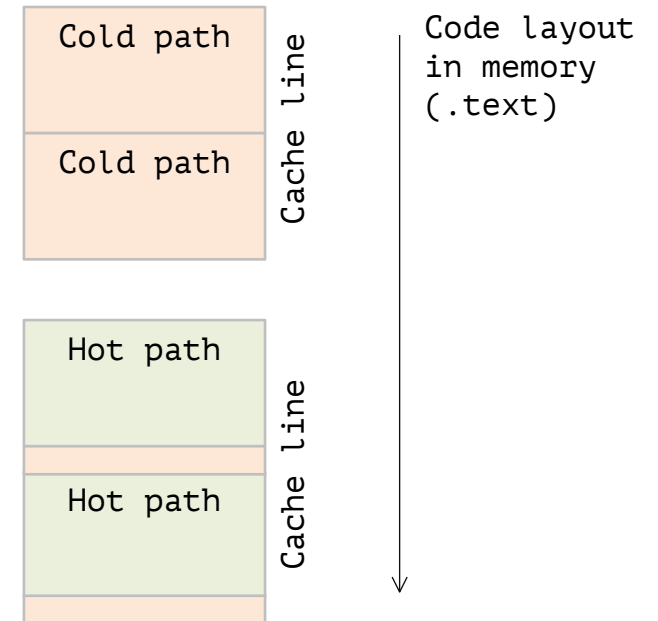
Разбиение больших функций (function splitting)

```
void process_error() __attribute__((noinline))
{ // Cold code ... }

void exit_error() __attribute__((noinline))
{ // Cold code ... }

void process(void *data)
{
    // Hot code path
    if (error)
        process_error();

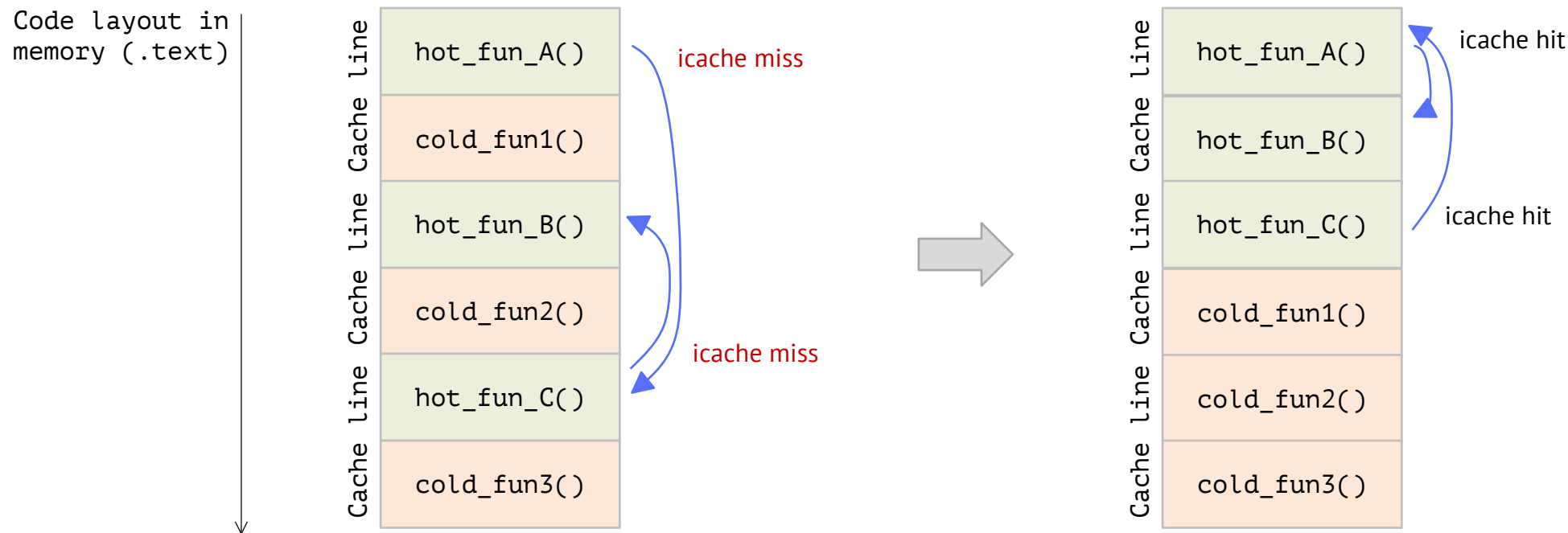
    // Hot code path
    if (error)
        exit_error();
}
```



- Блоки холодного кода вынесены в отдельные функции (встраивание функций отключено)
- Код функции process() в основном состоит из блоков горячего пути, что приводит к более эффективному использованию кеш-памяти инструкций (предвыборка, промахи)

Оптимизация размещения кода функций в памяти

- Небольшие функции с блоками кода на горячем пути можно объединить в одну функцию для более эффективного использования кеш-памяти инструкций (grouping)
- При компоновке функции с блоками кода на горячем пути можно разместить в объектном файле последовательно, чтобы сократить возможные промахи кеш-памяти инструкций (reordering)
 - LLVM LLD: `--symbol-ordering-file`
 - LLD поддерживает упорядочивание размещения функций по результатам профилирования (HFSort)
 - Profile-based relinking: BOLT, Google Propeller



Оптимизация размещения кода функций в памяти

```
void fun1(int *a, int n) {
    for (int i = 0; i < n; i++) {
        a[i] = 2 * a[i];
    }
}

int cold_fun1(uint32_t x) { ... }

void fun2(int *a, int n) {
    for (int i = 0; i < n; i++) {
        a[i] = a[i] * a[i];
    }
}

int cold_fun2(uint32_t x) { ... }

void fun3(int *a, int n) {
    for (int i = 0; i < n; i++) {
        a[i] = a[i] * a[i] * a[i];
    }
}

int cold_fun3(uint32_t x) { ... }
```

```
void comp(int *a, int n) {
    int x = 0;

    for (int i = 0; i < 100; i++) {
        fun1(a, n);
        x = cold_fun1(a[i]);

        fun2(a, n);
        x += cold_fun2(a[i]);

        fun3(a, n);
        x += cold_fun3(a[i]);
    }
    a[0] = x;
}
```

Оптимизация размещения кода функций в памяти

```
00000000000012c0 <fun1> size = 32B:
 12c0:  f3 0f 1e fa          endbr64
  ...
 12d9:  75 f5                jne    12d0 <fun1+0x10>
 12db:  c3                  ret
 12dc:  0f 1f 40 00         nopl   0x0(%rax)
00000000000012e0 <cold_fun1> size = 64B:
  ...
 131f:  c3                  ret
0000000000001320 <fun2> size = 48B:
 1320:  f3 0f 1e fa          endbr64
  ...
 133f:  75 ef                jne    1330 <fun2+0x10>
 1341:  c3                  ret
 1342:  66 66 2e 0f 1f 84 00 data16 cs nopw 0x0(%rax,%rax,1)
 1349:  00 00 00 00
 134d:  0f 1f 00            nopl   (%rax)
0000000000001350 <cold_fun2> size = 64B:
  ...
 138f:  c3                  ret
0000000000001390 <fun3> size = 48B:
 1390:  f3 0f 1e fa          endbr64
  ...
 13b4:  75 ea                jne    13a0 <fun3+0x10>
 13b6:  c3                  ret
 13b7:  66 0f 1f 84 00 00 00 nopw   0x0(%rax,%rax,1)
 13be:  00 00
00000000000013c0 <cold_fun3> size = 64B:
  ...
 13ff:  c3                  ret
```

адрес 0x12c0 % 64 == 0, выравнен на границу строки L1i

L1i cache line 64B:



адрес 0x12e0 % 8 == 0, выравнен на границу 8 байт

адрес 0x1320 % 8 == 0, выравнен на границу 8 байт

Литература

- Denis Bakhvalov. **Performance Analysis and Tuning on Modern CPUs: Squeeze the last bit of performance from your application** // easyperf.net
- Aart J.C. Bik. **Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance**
- Randal E. Bryant, David R. O'Hallaron. **Computer Systems: A Programmer's Perspective**
- **Intel Intrinsic Guide** // <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/>