Московский физико-технический институт
(государственный университет)
Физтех-школа радиотехники и компьютерных технологий
Кафедра информатики и вычислительной техники

Повышение производительности обработки сетевых пакетов с помощью JIT-компиляции Berkeley Packet Filter в ядре операционной системы Эльбрус

Выпускная квалификационная работа
(магистерская диссертация)

Выполнил: Михайлов К.Н.

Научный руководитель: к.т.н. Жмурин А.В.

Консультант: Фёдоров А.В.

Введение 1. Предпосылки для ускорения обработки сетевых пакетов на машинах архитектуры Эльбрус

- При запусках теста на производительность сети netperf на машинах архитектуры Эльбрус наблюдалась 100% загрузка процессора. Процессор загружен обработкой сетевых пакетов, которую выполняет ядро операционной системы.
- Среди прочих задач около 6% системного времени теста занимала исполнение программы Berkeley Packet Filter (BPF).
- Необходимо ускорить исполнение программ BPF для ускорения обработки сетевых пакетов.

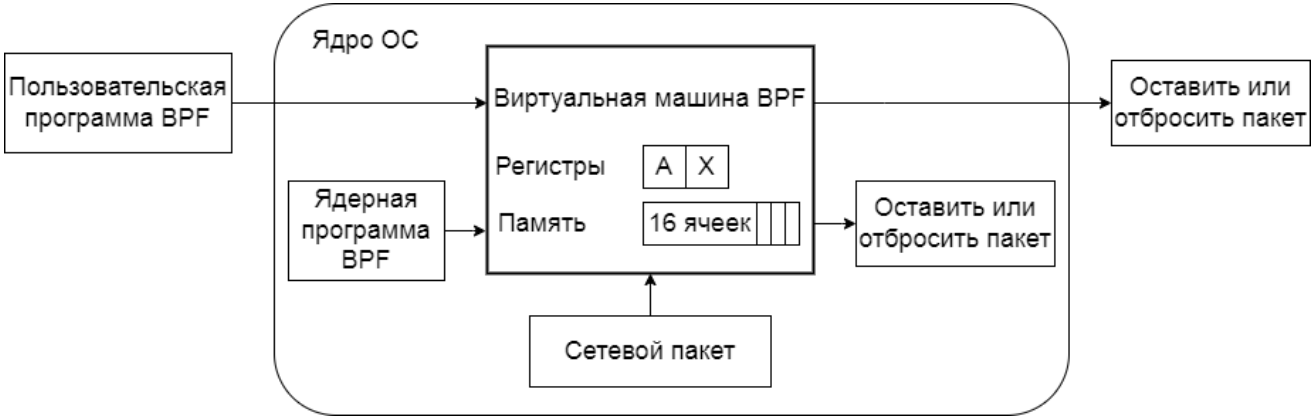
Введение 2. Фильтрация сетевого трафика при помощи фильтров BPF

- Фильтрация на уровне ядра:

- Загрузка фильтра в ядро – через системный вызов setsockopt().
- Фильтр вызывается после копирования пакета в буфер ядра. Если пакет не нужен, он не будет копироваться пользовательскому приложению.
- Также ядро Linux имеет несколько фильтров в своём составе, например, фильтр для пакетов протокола PTP.

Введение 3. Описание BPF

- Berkeley Packet Filter (BPF) – технология, позволяющая запускать программы на специальном языке в контексте ядра ОС. Используется для работы с сетевыми пакетами.
- Программы BPF исполняются на виртуальной машине BPF, которую эмулирует ядро операционной системы:



- Программы BPF представляют собой последовательность инструкций из набора команд BPF.

Введение 4. Набор команд BPF

Группа инструкций	Количество	Пример
Арифметические и логические	22	A += X
Переход внутри программы	9	pc += (A == X) ? 2 : 8
Возврат из программы	2	return A
Работа с памятью BPF	4	st A, mem[10]
Работа с данными сетевых пакетов	24	ld data[200], A
Пересылка между регистрами	2	X = A
Загрузка констант в регистры	2	A = 5

Введение 5. JIT-компилятор для BPF

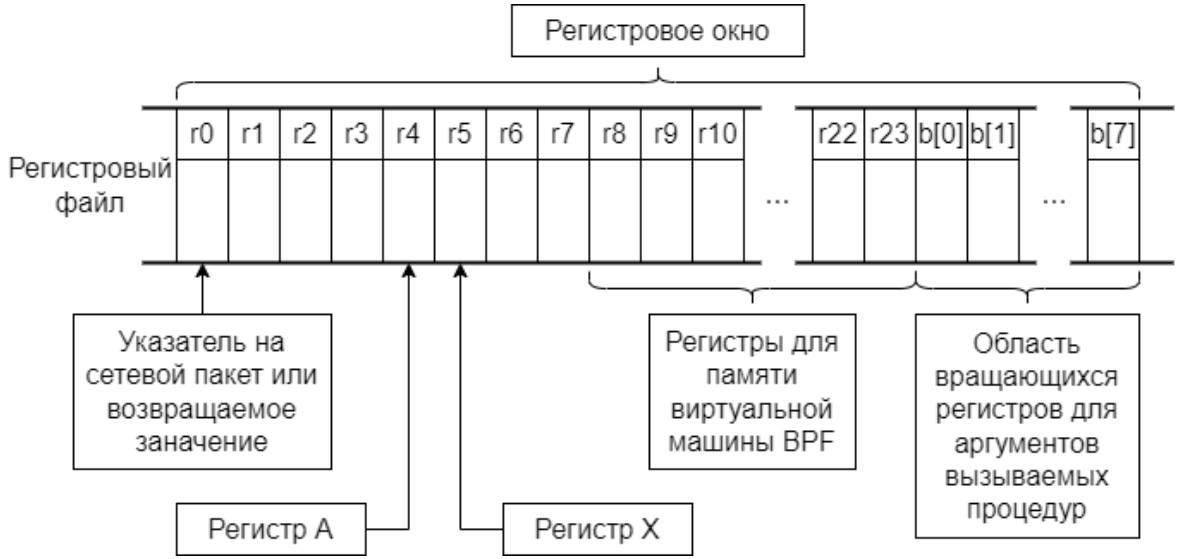
- Универсальный способ исполнения BPF программ в ядре Linux – интерпретация. Однако интерпретация медленно работает на Out-of-Order процессорах и особенно на процессорах со статическим планированием, таких как Эльбрус.
- Для ускорения исполнения BPF-программ в ядре Linux было предложено компилировать программы напрямую в машинный код. При таком подхода выигрывают и Out-of-Order процессоры, и особенно Эльбрус.
- Инструмент для преобразования BPF-программы в машинный код получил название Just-in-Time компилятор BPF (сокр. BPF JIT). Он должен находиться в архитектурно-зависимой части ядра.

Цели и задачи

- Цель работы – ускорить исполнение BPF-программ на архитектуре Эльбрус с помощью JIT-компиляции BPF.

- Задачи:
  1) Разработать схему эмуляции виртуальной машины BPF на основе аппаратных возможностей Эльбруса;
  2) Подготовить шаблоны инструкций BPF на языке ассемблера;
  3) Разработать и реализовать алгоритм JIT-компиляции программ BPF;
  4) Оптимизировать шаблоны под различные версии архитектуры Эльбрус;
  5) Встроить JIT-компилятор в ядро ОС Эльбрус;
  6) Измерить ускорение работы BPF-программ с JIT-компиляцией.

Схема эмуляции виртуальной машины BPF на основе аппаратных возможностей Эльбруса



- Регистры под память BPF и под аргументы вызываемых функций выделяются только при необходимости.
- JIT-компилятор на основе программы BPF строит инициализацию регистрового окна в прологе функции. Она состоит из:
  - Указания размера регистрового окна;
  - Определения область вращающихся регистров.

Шаблоны инструкций BPF на языке ассемблера

Группы широких команд на ассемблере Эльбруса были объединены в шаблоны:

- Каждый шаблон соответствует инструкции BPF;
- Шаблоны компилируются вместе с ядром, поэтому JIT оперирует объектными кодами шаблонов;
- Пример шаблона для инструкции A %= X:

```
SYM_FUNC_START_SIZED(bpf_jit_alu_mod_x)
{
        sxt,5 0x6, %r_A, %dr_A
}
{
        umodx,5 %dr_A, %r_X, %r_A
}
SYM_FUNC_END_SIZED(bpf_jit_alu_mod_x)
```

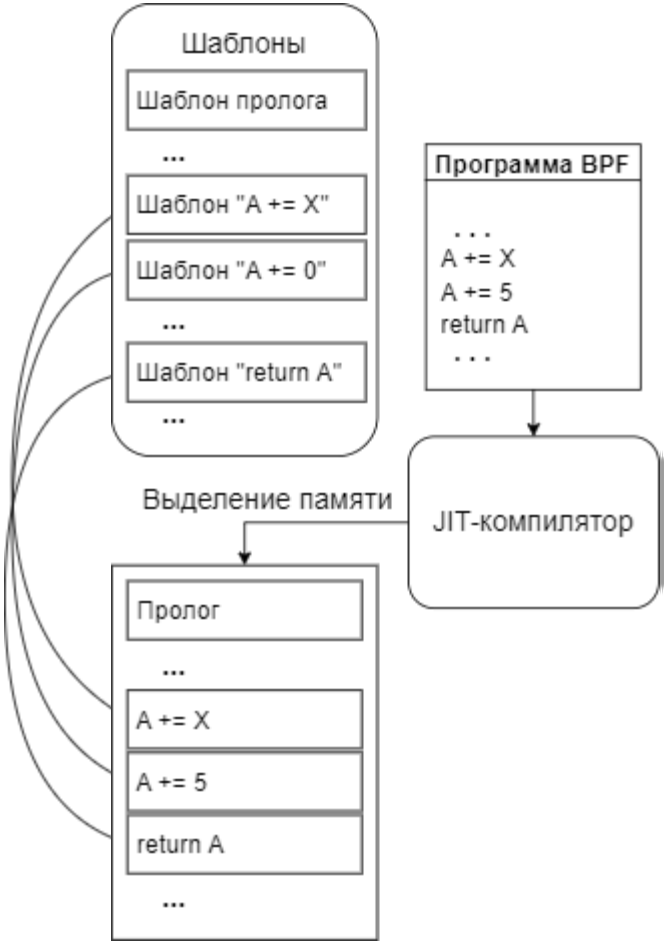
Алгоритм JIT-компиляции

Входные данные: BPF-программа.

Алгоритм JIT-компиляции:

1) Для каждой инструкции исходной программы:
   определить размер соответствующего шаблона;
   отметить используемые инструкцией ресурсы;
2) Выделить память под компилируемую программу. Размер памяти - сумма длин шаблонов;
3) Вставить пролог с выделением нужных ресурсов;
4) Для каждой инструкции исходной программы сформировать объектный код на основе шаблона в выделенной памяти.

Результат: скомпилированная программа в выделенной памяти.



Реализация алгоритма JIT-компиляции
Формирование объектного кода на основе шаблонов

Места результирующего объектного кода, в которых он отличается от объектного кода шаблона:

- константы из инструкций BPF;
- смещение при переходах;
- смещение до вызываемых функций и базу их регистрового окна;
- размер окна текущей процедуры;
- базу вращающихся регистров.

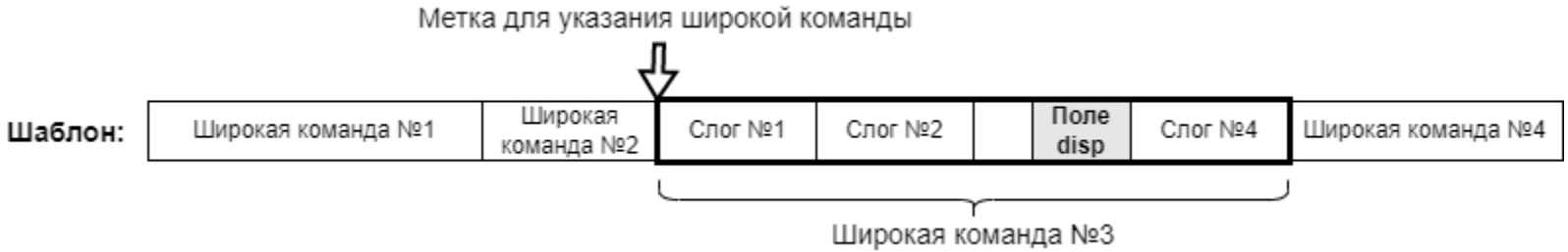
Чтобы определить эти места, JIT должен последовательно определить смещения:

1. широкой команды в шаблоне;
2. слога в широкой команде;
3. поля в слоге.

Реализация алгоритма JIT-компиляции

Формирование объектного кода на основе шаблонов (продолжение)

1. Для определения смещения широкой команды от начала шаблона был разработан подход с использованием меток в шаблоне. Метка должна стоять в коде шаблона перед нужной широкой командой. Имя метки известно JIT-компилятору.



2. Для определения сдвига слога от начала широкой команды использовались функции из архитектурно-зависимой части ядра ОС Эльбрус. Части нужных функций в ядре не было и их пришлось реализовать.
3. Для определения сдвига поля от начала слога использовались структуры слогов из архитектурно-зависимой части ядра ОС Эльбрус. Части нужных структур в ядре не было и их пришлось реализовать.

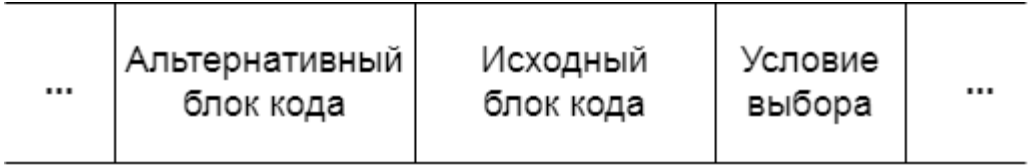
Оптимизация шаблонов под различные версии архитектуры Эльбрус
Описание проблемы

- JIT-компилятор не может переставлять инструкции между шаблонами, поэтому каждый шаблон имеет максимальную параллельность инструкций в широких командах.
- Для этого нужно по максимуму наполнять широкие команды шаблона, при этом учитывая все задержки между инструкциями.
- Однако задержки в разных версиях архитектуры Эльбрус разные. Например, чтение из L1 до шестой версии требовало задержку 3 такта, а начиная с шестой – 5 тактов.
- Требуется сделать шаблоны оптимальными для исполнения на процессоре любой версии архитектуры Эльбрус.

Оптимизация шаблонов под различные версии архитектуры Эльбрус

Решение проблемы путём применения механизма альтернатив

- Альтернативы – механизм ядра ОС Эльбрус, позволяющий ядру исполнять один из двух или трёх заранее подготовленных различных блоков кода.



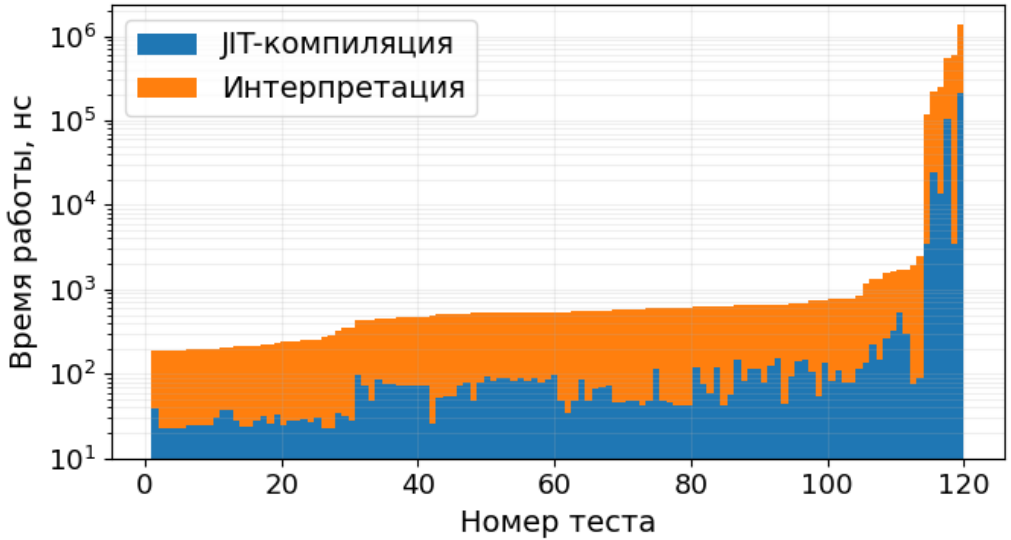
- Условие выбора блока кода – динамическое: ядро при инициализации проверяет условие выбора и подставляет нужный блок кода.
- Следующие блоки кода в шаблонах были реализованы с использованием механизма альтернатив:
  - обращение в память;
  - выработка предикатов для арифметических операций и для операций передачи управления.
- Например, для учёта изменения задержки при чтении из памяти (предполагается попадание в L1) в шаблоне должны быть два альтернативных блока кода: один – для версии архитектуры до v6 и другой – для v6 или выше.

Включение JIT-компилятора в ядро ОС Эльбрус

- JIT-компилятор был реализован в виде архитектурно-зависимой функции bpf_jit_compile().
- Для выделения памяти под компилируемую программу вызывается специальный аллокатор – bpf_jit_binary_alloc(). Он помечает память как исполняемую и создаёт для скомпилированной программы дополнительную защиту от выхода за границы.
- Для запуска BPF-программы вызывается скомпилированная функция.

Ускорение работы BPF-программ с JIT-компиляцией

- Измерения проводились на тестах из ядерного модуля test_bpf. Он содержит порядка 120 тестов на все возможные инструкции BPF. Сравнивалась скорость работы тестов без JIT-компилятора и с ним. В среднем исполнение ускорилось в 10,4 раза.



- Также была проведена эмуляция запуска программы BPF, которая занимала около 6% процессорного времени. Она ускорилась в 5,5 раз.

Результаты

1) Разработана схема эмуляции виртуальной машины BPF на основе аппаратных возможностей Эльбруса;
2) Реализованы 67 шаблонов на ассемблере;
3) Разработан алгоритм JIT-компиляции программ BPF и реализован JIT-компилятор на его основе;
4) Шаблоны оптимизированы под различные версии архитектуры Эльбрус;
5) JIT-компилятор добавлен в ядро ОС Эльбрус версии 6.1;
6) JIT-компиляция увеличила скорость работы BPF-программ из набора тестов test_bpf более чем в 10 раз.

Дополнительная информация

Пример использования

- Фильтрация пакетов IPv4 на интерфейсе eth0:

```
$ tcpdump -i eth0 ip
```

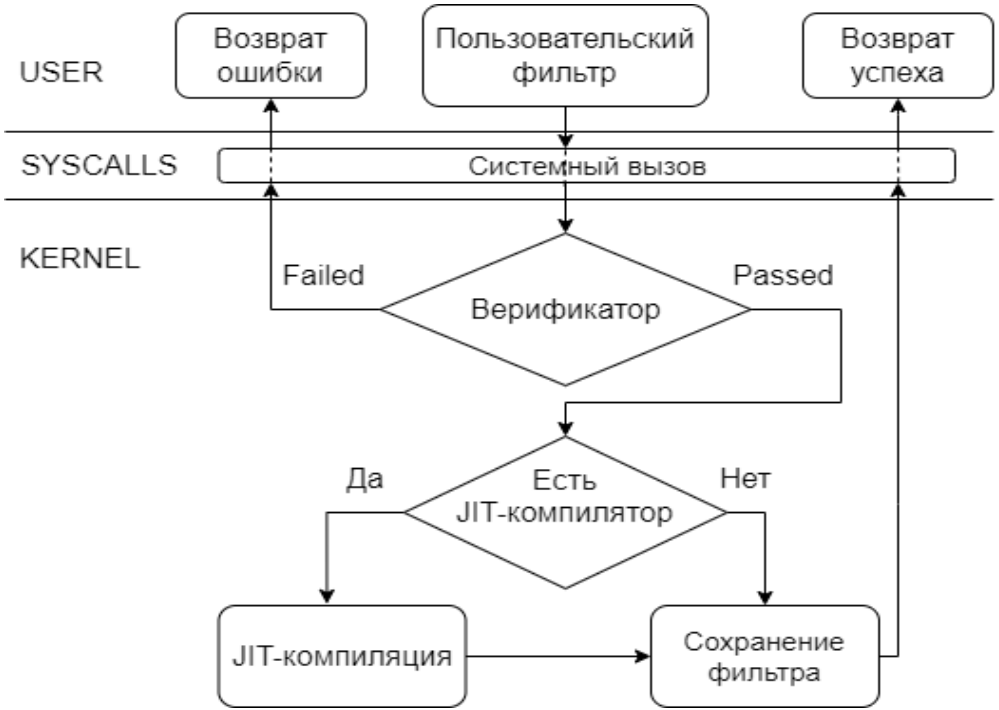
- Сгенерированный этой командой BPF-фильтр:

```
(000) ldh [12]
(001) jeq      #0x800     jt 2    jf 3
(002) ret      #262144
(003) ret      #0
```

Применения BPF

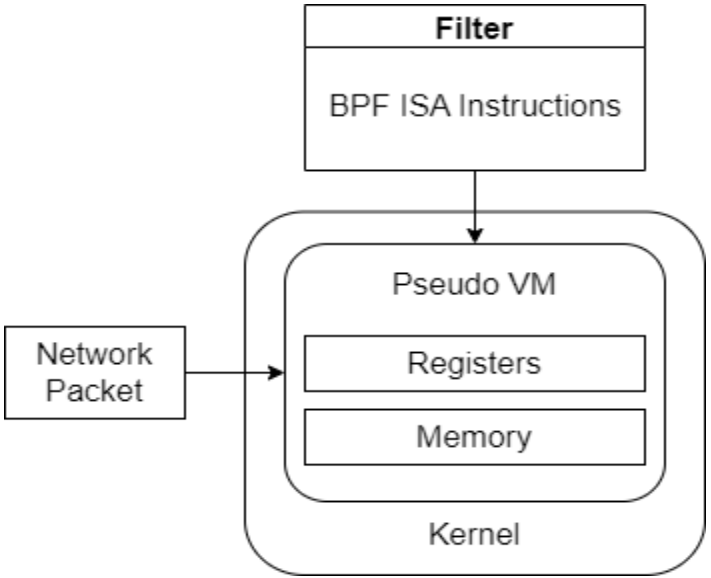
- Изначальное применение BPF – фильтрация сетевых пакетов.
- Со временем возможность исполнять произвольные программы в контексте ядра получила признание и дальнейшее развитие.
- Новыми применениями стали:
  - задачи трассировки и профилирования;
  - механизм безопасного выполнения процессов (seccomp).
- Использование BPF ядром – фильтр пакетов протокола PTP для синхронизации времени.

Загрузка пользовательского фильтра в ядро ОС

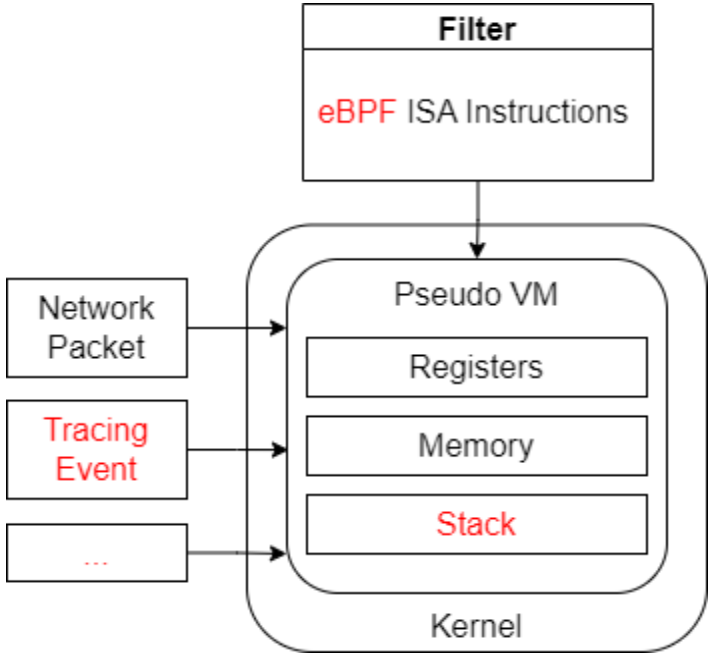


Виртуальная машина BPF

Classic BPF



Extended BPF



JIT-компиляция BPF (1 из 2)

До компиляции:

```
/* A += X */
{0x0c, 0, 0, 0}

/* return 10 */
{0x06, 0, 0, 10}

/* pc += (A == X) ? 2 : 3 */
{0x1d, 2, 3, 0}
```

JIT-компиляция BPF (2 из 2)

До компиляции:

```
/* A += X */
{0x0c, 0, 0, 0}
```

После компиляции:

```
/* A is r4, X is r5 */
adds,0 %r4, %r5, %r4
```

На самом деле:

```
/* HS */      0x04000001
/* ALS0 */    0x10848584
```

JIT первой итерации

```
case BPF_ALU | BPF_DIV | BPF_X: /* A /= X */
        /* if X is 0 return 0 */
        if (cmd_len_and_break(pass, cmd_lengths, i, 5))
                break;
        ptr[pc++] =     /* HS */        0x04005093; // als[0] = 1, nop = 1, lng = 1, mdl = 3
        ptr[pc++] =     /* SS */        0x80000000; // ipd 2
        ptr[pc++] =     /* ALS0 */      0x2089c040; // cmpesb,0 %r9, 0x0, %pred0
        ptr[pc++] =     /* CS0 */       0xf0000000; // return %ctpr3

        ptr[pc++] =     /* HS */        0x84010112; // als[5] = 1, als[0] = 1, cd = 1, nop = 2, lng = 1, mdl = 2
        ptr[pc++] =     /* ALS0 */      0x10c0c080; // adds,0 0x0, 0x0, %r0 ? %pred0
        ptr[pc++] =     /* ALS5 */      0x40888988; // udivs,5 %r8, %r9, %r8 ? ~ %pred0
        ptr[pc++] =     /* CDS0 */      0x52600460; // rlp,cd00 %pred0, ~>alc5
                                                    // rlp,cd01 %pred0, >alc0

        ptr[pc++] =     /* HS */        0x00001001; //
        ptr[pc++] =     /* SS */        0xc0000c40; // ct %ctpr3 ? pred0
                                                    // ipd 3
        break;
```

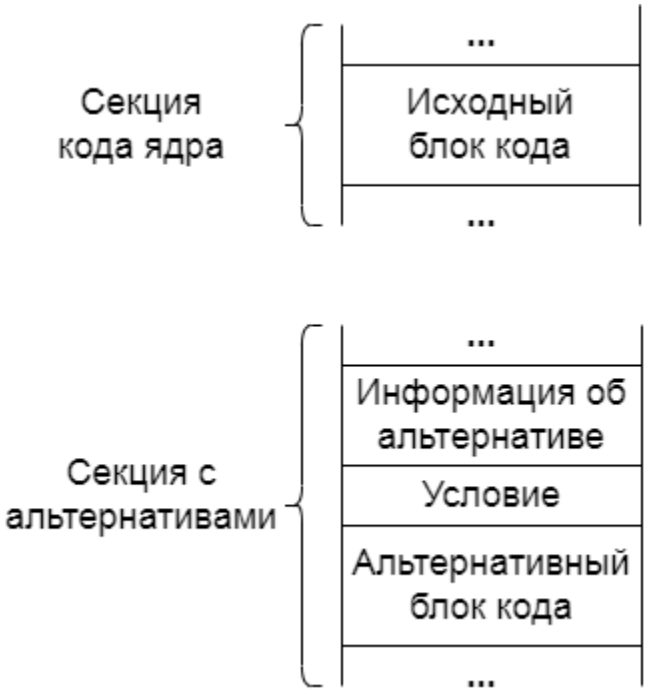
Пример использования метки в шаблоне

- Шаблон для инструкции A %= Const:

```
SYM_FUNC_START_SIZED(bpf_jit_alu_mod_k)
{
        sxt,5 0x6, %r_A, %dr_A
}
BPF_TEMPL_LABEL(BPF_JIT_K_LABEL(bpf_jit_alu_mod_k))
{
        umodx,5 %dr_A, _f32s,_lts0 0x0, %r_A
}
SYM_FUNC_END_SIZED(bpf_jit_alu_mod_k)
```

Устройство альтернатив

Альтернатива – возможность исполнять один из двух или трёх различных блоков кода.



Альтернативные блоки кода хранятся в специальной секции.

- При инициализации ядро проходит по всей секции альтернатив.
- Для каждой альтернативы вычисляется условие; если оно верно, то альтернативный блок кода вставляется вместо исходного.

Пример использования альтернативы в шаблоне

```
SYM_FUNC_START_SIZED(bpf_jit_ld_w_len)
ALTERNATIVE_1_ALTINSTR
        {
                nop 4
                ldw,0 [ %dr0 + _f32s,_lts0 SKB_LEN ], %r_A
        }
ALTERNATIVE_2_OLDINSTR
        {
                nop 2
                ldw,0 [ %dr0 + _f32s,_lts0 SKB_LEN ], %r_A
        }
ALTERNATIVE_3_FEATURE(CPU_FEAT_ISET_V6)
SYM_FUNC_END_SIZED(bpf_jit_ld_w_len)
```

Ускорение работы BPF-программ с JIT-компиляцией

- Измерения проводились на тестах из ядерного модуля test_bpf. Он содержит порядка 120 тестов на все возможные инструкции BPF.
- Сравнивалась скорость работы тестов без JIT-компилятора и с ним:

Интерпретация:

```
#0 TAX jited:0 1115 1088 1087
#1 TXA jited:0 363 362 363
#2 ADD_SUB_MUL_K jited:0 473
#3 DIV_MOD_KX jited:0 1094
...
```

JIT-компиляция:

```
#0 TAX jited:1 127 126 152
#1 TXA jited:1 40 40 40
#2 ADD_SUB_MUL_K jited:1 50
#3 DIV_MOD_KX jited:1 160
...
```

- В среднем исполнение ускорилось в 10,4 раза.

Ускорение работы BPF-программ с JIT-компиляцией

- На E8CB исполнение BPF-программ ускорилось в 10,3 раз.