Reuse Host JIT Back-end as Offload Back-end

Jiong Wang
Netronome
LPC BPF Microconference, September-11, 2019
Agenda

- Why do we need this?
- BPF prog to runnable image
  - Host JIT without BPF2BPF call
  - Host JIT with BPF2BPF call
  - Static offload JIT
  - Dynamic offload JIT
- JIT back-end improvements for better modularity
  - Enable multiple JIT back-ends at the same time
  - Code-gen only and PIC code only
  - Separate compilation and linking
- Prototyping based on SmartNIC with RISC-V inside
  - Hardware introduction
  - Software prototyping
Why do we need this?

- BPF could be offloaded
  - SmartNIC at the moment (Netronome NFP)
  - Perhaps other devices in the future once device driver on BPF
- And we want to use architectures with strong ecosystem
  - RISC-V, arm32, AArch64 etc or even BPF itself[1]
  - There are host JIT back-ends for them already
- Host JIT and offload JIT
  - No difference on processor, code generation is the same
  - Difference on runtime environment, linking is different
  - We want to reuse code generation part of host JIT

[1]: “Programmable Dataplane for Next Generation Networks”, Glasgow University.
BPF Prog -> Runnable Image

- A runnable image must have:
  - All BPF instructions translated
  - All external references relocated (Maps, branches, calls)
  - Hence, BPF prog must have the followings resolved during JIT compilation:
    - Map addresses (static global data is based on map as well)
    - Destinations of local jumps, helper calls, BPF2BPF calls

- Current BPF JIT infrastructure
  - Resolving them on BPF ISA instead of native ISA
    - C -> relocatable BPF .o -> final BPF.o -> final image (Selected)
    - C -> relocatable BPF .o -> relocatable native image -> final image (Not)
  - Two main stages for JIT compilation
    - prelinking BPF .o
    - JIT back-end code generation
BPF Prog -> Runnable Image

- Host JIT - prelinking without BPF2BPF call

**User Space**

- Loader
  - alu:
    - r0 += 2
    - r1 = map_obj
  - jmp:
    - if r0 == 4 goto -alu
    - if r0 == 6 goto +call
  - call:
    - call helper_idx

**Kernel Space**

- Kernel space "loader"
  - alu:
    - r0 += 2
    - r1 = map_idx
  - jmp:
    - if r0 == 4 goto -alu
    - if r0 == 6 goto +call
  - call:
    - call helper_idx

- Kernel space verifier
  - map/ helper_call rewriter
  - map

- Helper functions
  - maps

- BPF Prog
  - map/rewriter

- Kernel space verifier
  - pc-rel call helper_addr

- Only PROGBITS (insn/data) are loaded into kernel space
- Reloc and symtab sections etc won’t be loaded
- Reloc info therefore needs to be re-encoded into insn

- BPF sequence have all external **address finalized before** JIT code generation
BPF Prog -> Runnable Image

- Host JIT - prelinking with BPF2BPF call
  - C -> relocatable BPF .o -> final BPF.o -> native image, this flow has dilemma

Kernel Space

- Still, BPF sequence has all address finalized including subprogs before JIT code generation
BPF Prog -> Runnable Image

- Host JIT - code generation
  - Input: BPF sequence has all address finalized including subprogs before JIT code generation

Kernel Space
BPF Prog -> Runnable Image

● Summary for host JIT
  ○ User and kernel space loaders perform various prelinkings on BPF ISA
    ■ Three “symbol” tables used:
      ● map_idx -> map_addr
      ● helper_idx -> helper_addr
      ● func_idx -> func_addr
  ○ JIT back-ends **interleave with prelinking** because of the flow dilemma
  ○ Some JIT back-ends generates **non-PIC** instruction sequence in final image
  ○ No relocation information in final image
BPF Offload - Static Offload

- Data/Code (maps, prog, libs/helpers) preallocated on devices
- Extern addresses for BPF prog still could be known before doing JIT code-gen
- Prelinking on BPF ISA then doing code-gen still work
- `replace_map_fd_with_map_ptr` and `fixup_bpf_call` need tweaks

Diagram:

- User space
  - Loader
  - PROG_LOAD
  - Insn rewriter
  - JIT back-end

- Kernel space
  - Image ready to run on device

- Offload Device
  - `map` (created on device for MAP_CREATE syscall)
  - `lib/helpers` (pre-installed on device)
  - `code` (preallocated from fixed address)
BPF Offload - Dynamic Offload

- Code could have been allocated dynamically on devices
- If code generation uses PIC sequence, then no difference with static offload
- Otherwise, needs runtime relocation information

Diagram:

- User space: Loader → PROG_LOAD → Insn rewriter → JIT back-end
- Kernel space: map (created on device for MAP_CREATE syscall)
- Offload Device: lib/helpers (pre-installed on device)
- Code (allocated by loader from random address)
- Image ready to run on device

For non-PIC call sequence, needed but unknown
• For example, NFP doesn’t support pc-relative jump/call, we have the following dynamic offload implementation:

```assembly
alu:
  r0 += 2
  r1 = map_idx (actually fd)
call helper_idx
jmp:
  if r0 == 4 goto -alu
  if r0 == 6 goto +call
call:
  call helper_idx
```

Map index and helper index are kept as symbol index for runtime relocation done by loader on SmartNIC

→ No hardware pc-relative jump, so all jumps needs R_REL to adjust the jump destination according to load base

→ A few special relocation, for example exit point

→ Relocation value is not splitted into sequence

→ NFP insn is 64-bit, but a few top bits are reserved, so relocation types are kept there!
BPF Offload - Summary

● The current host JIT back-ends could perhaps be used as offload JIT back-ends directly with very little changes, because:
  ○ Native data (maps) and code are created separately. We always know data addresses before generating code
  ○ The generated code themselves could be PIC (Position Independent Code)
  ○ Offload JIT may need to generate extra runtime code
    ■ return from main returns to other device firmware exit
    ■ error handling code
    ■ device could expose these addresses to offload JIT

● If not
  ○ The offloaded image needs to encode the relocation information, perhaps the offload image needs an extra header
JIT Back-end Improvements

- Enable multiple JIT back-ends at the same time
  - x86_64/AArch64 + offload device 1(Arm) + offload device 2(RISC-V) ... etc. could be the usual architecture combination
  - We need multiple JIT back-ends enabled, not only the $(ARCH)
    - JIT back-end normally is a single file, could be built independently

- Solution
  - Split bpf_int_jit_compile into bpf_int_jit_compile + ARCH_bpf_int_jit_compile
  - bpf/core.c defines a set of weak ARCH_bpf_int_jit_compile for all
  - Extra interface to query what’s the offload arch
JIT Back-end Improvements

- Enable multiple JIT back-ends at the same time - no offload

```c
bpf_int_jit_compile_all[] =
{
    x86_64_bpf_int_jit_compile,
    riscv_bpf_int_jit_compile,
    ...
}
```

```c
bpf_jit_needs_zext_all(enum jit_arch)
{
    case X86_64:
        return false;
    case RISCV:
        return true;
    case ...
}
```

```c
bpf_jit_interface.c:
    bpf_int_jit_compile()
    {
        x86_64_bpf_int_jit_compile();
    }
```

```c
bpf_jit_backend.c
    x86_64_bpf_int_jit_compile()
    {
        the implementation...
    }
```

kernel/bpf/core.c

Host arch interface overrides the weak interface
JIT Back-end Improvements

- Enable multiple JIT back-ends at the same time - with offload

```c
bpf_int_jit_compile() ((__weak__))
bpt_jit_needs_zext() (__weak__))

bpf_int_jit_compile_all[] =
{
    x86_64_bpf_int_jit_compile,
    riscv_bpf_int_jit_compile,
    ...
}

bpf_jit_needs_zext_all(enum jit_arch)
{
    case X86_64:
        return false;
    case RISCV:
        return true;
    case ...
}

x86_64_bpf_int_jit_compile() ((__weak__))
riscv_bpf_int_jit_compile() ((__weak__))
...
```

Host arch interface overrides the weak interface

```c
bpf_jit_interface.c:
bpf_int_jit_compile(){
    x86_64_bpf_int_jit_compile();
}

bpf_jit_backend.c
x86_64_bpf_int_jit_compile() {
    the implementation...
}
```

```c
kernel/bpf/core.c
```

```c
kernel/bpf/Makfile:
obj-$(CONFIG_BPF_JIT_BACKEND_RISCV) = $(objtree)/../arch/$(SRCARCH/net/bpf_jit_backend.c
```

```c
arch/x86/net/
```

```c
bpf_jit_interface.c:
bpf_int_jit_compile(){
    riscv_bpf_int_jit_compile();
}

bpf_jit_backend.c
riscv_bpf_int_jit_compile() {
    the implementation...
}
```

```c
arch/riscv/net/
```
JIT Back-end Improvements

- Cleaner code generation
  - Back-end generates PIC code as much as possible when range fits
  - Back-end does code-gen only, no runtime stuff (icache flush)
  - Split compilation and linking?
    - `bpf_int_jit_compile()`
    - `bpt_int_jit_link(bpf_prog, Idx2Addr map, Idx2Addr helper, Idx2Addr subprog)`
      - More reloks compared with BPF ISA. Arches could split reloc value into sequence for loading large imm. `mov r0, addr_0_16, movsh r0, addr_16_32, movsh r0, addr_32_48, movsh r0, addr_48_64`
      - Architecture has their own relocation description, for example `R_AARCH64_MOVW_*`, `R_RISCV_HI_*` etc.
      - Pro is no need of back-end dry run inside verifier
      - Con is more back-ends related work.
Offload Infrastructure Improvements

- Offload JIT is bypassing a couple of paths of host JIT
  - Designed for NFP offload
  - Could be overkilling for generic RISC processors offload
  - For example, we could still want prelinking on BPF ISA

- Current offload infrastructure was more or less designed for net devices, may could be simplified for other offload scenarios.
Hardware and Software Prototyping

- Netronome RFPC (RISC-V Flow Processing Core)

- The chip or chiplet is made up of islands, which are connected through the instruction-driven switch fabric
- Which allows for implementation from small to large
- Memory hierarchy provides equal access to all types of memories
- The config, host interface, and network interface islands allow for feeding data into the system
- Basic flow of data in a SmartNIC
Hardware and Software Prototyping

- Netronome RFPC - continues
RFPC (RISC-V Flow Processing Core) features:

- RFPC cores are RV32IMC cores with custom-0/1 instructions
  - RV32IMC keeps the performance high with low silicon gate count
  - Support for user, machine and debug modes only, but provides some memory protection and both user-level and machine-level interrupts
  - Custom-0 instructions permit dynamic binding of 48+-bit host address and bulk DDR addresses to 32-bit RISC-V addresses
  - Custom-1 instructions permit transaction memory and signaling operations

- RFPC Cores collected into RFPC groups
  - Sharing local memory, which is directly accessed (not cache)
  - Simple address translation permits core-local data and stack without changing code and register initialization values

- RFPC Groups collected into RFPC Clusters
- RFPC Clusters collected together
Software prototyping - basic environment rough description

- Standard C program
- RISC-V ELF .o
- riscv-elf-gdb/run

Arm Controller
- gdb-remote-stub

RFPC cores on FPGA

x86 host \(\leftrightarrow\) remote FPGA board

net
Hardware and Software Prototyping

- **Software prototyping - BPF offload, crazy ideas**

  - `bpf.o` → `libbpf` → `dummy netdev driver` → `bpf_prog`
  - `x86 host` → `redirect map_create` → `remote FPGA board`
  - `libbpf` → `bpftool`
  - `raw insn bin` → `convertor` → `loadable bin`
  - `bpf_prog` → `bpftool`
  - `redirect map_create` → `gdb-remote-stub`
  - `pkt gen verif` → `inject code bin` → `RFPC code`
  - `loadable bin` → `offloaded bpf code` → `code polling event`
  - `RFPC data`

  : kernel space
Thank You