Efficient JIT to 32-bit Arches

Jiong Wang

Linux Plumbers Conference
Vancouver, Nov, 2018

Background

NETRONOME

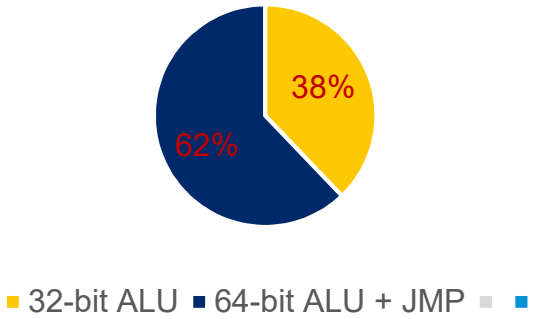
- ISA specification and impact on JIT compiler
  - Default code-gen use 64-bit register, ALU64, JMP64
  - No big impact on 64-bit backends (REX header for x86-64)
  - **Requires extra regs and insns on 32-bit arches.**
- Programs in real world are 32-bit friendly
  - Nearly all non-pointer arithmetic are 32-bit and all will be if pointer is 32-bit

Arch	BPF_ALU/BPF_JMP

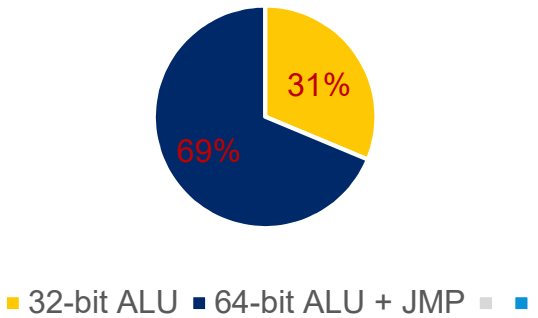
arm	emit_alu_r(rd[1], rs[1], true, false, op, ctx);
	emit_alu_r(rd[0], rs[0], true, true, op, ctx)
	
	emit(ARM_CMP_R(rd, rm), ctx);
	emit(ARM_COND_EQ, ARM_CMP_R(rt, rn), ctx);

nfp	emit_alu(…reg_both(dst), reg_a(dst), alu_op, reg_b(src)); emit_alu(…reg_both(dst + 1), reg_a(dst + 1), alu_op, reg_b(src + 1)); …

test_l4lb_noinline.c



test_xdp_noinline.c



Solution A - 32-bit subreg ISA

NETRONOME

- eBPF ISA already defined 32-bit subreg and ALU32 insns, could pass 32-bit semantics from c types down to assembly

```
void cal(u32 *a, u32 *b, u32 *c) {
  u32 sum = *a + *b;
  *c = sum;
}
```

default

```
cal:
  r1 = *(u32 *)(r1 + 0)
  r2 = *(u32 *)(r2 + 0)
  r2 += r1
  *(u32 *)(r3 + 0) = r2
  exit
```

-mattr=+alu32 (since LLVM 7.0)

```
cal:
  w1 = *(u32 *)(r1 + 0)
  w2 = *(u32 *)(r2 + 0)
  w2 += w1
  *(u32 *)(r3 + 0) = w2
  exit
```

- A subreg read is always 32-bit read, but write **implicitly zeros high 32-bit**. Compilers or hand written assembly could be using this

```
void cal(u32 *a, u32 *b, u64 *c) {
  u32 sum = *a + *b;
  *c = sum;
}
```

-mattr=+alu32

```
cal:
  w1 = *(u32 *)(r1 + 0)
  w2 = *(u32 *)(r2 + 0)
  w2 += w1
  *(u64 *)(r3 + 0) = r2
  exit
```

ALU32 insn but there is exploit of implicit zero extension on w2 by the following store

Reference a 32-bit subreg as a 64-bit definition

- 32-bit arches **must model implicitly zero extension** using extra insns to guarantee correctness. This applies to all ALU32 insn. How could we avoid these extra code-gen?

Solution A - 32-bit subreg ISA

NETRONOME

- Just don't do zero extension on the DST_REG of ALU32 at all, do them the first time DST_REG is used as 64-bit. Mark the reg to save on later use
- Insns have 64-bit register read including: cond BPF_JMP, BPF_ALU, BPF_STX | BPF_DW
  - cond JMP overhead could be reduced by introducing BPF_JMP32 insn. X86_64 and AArch64 ISA do support this

Benchmark	Total insn	Total cond JMP	JMP32	Percentage
test_xdp_noinline	999	84	52	61.9%
test_l4lb_noinline	569	40	24	60.0%

  - A large portion of BPF_ALU comes from address calculation, for example calculating address of local variables. Could potentially avoid this using verifier register type info
- Without flow analysis, when a instruction is jump destination (start of basic block), need to clear register mark. This could possibly cause quite a few unnecessary code-gen

Solution A - 32-bit subreg ISA

NETRONOME

- LLVM might have better knowledge on all these, so another choice is stopping LLVM exploiting implicit zero extension and generate it explicitly
- For explicit zero extension, we can’t use existing ALU32 MOV as otherwise we can’t differentiate it with normally MOV which we don’t want to clear high 32bits. We will need new explicit zero-extension instruction BPF_UEXT
- But how could verifier safely know the input sequences is compiled from LLVM compiler conforming to such convention is an issue. Could generate ELF tag, however which could be faked, therefore have potential impact on security

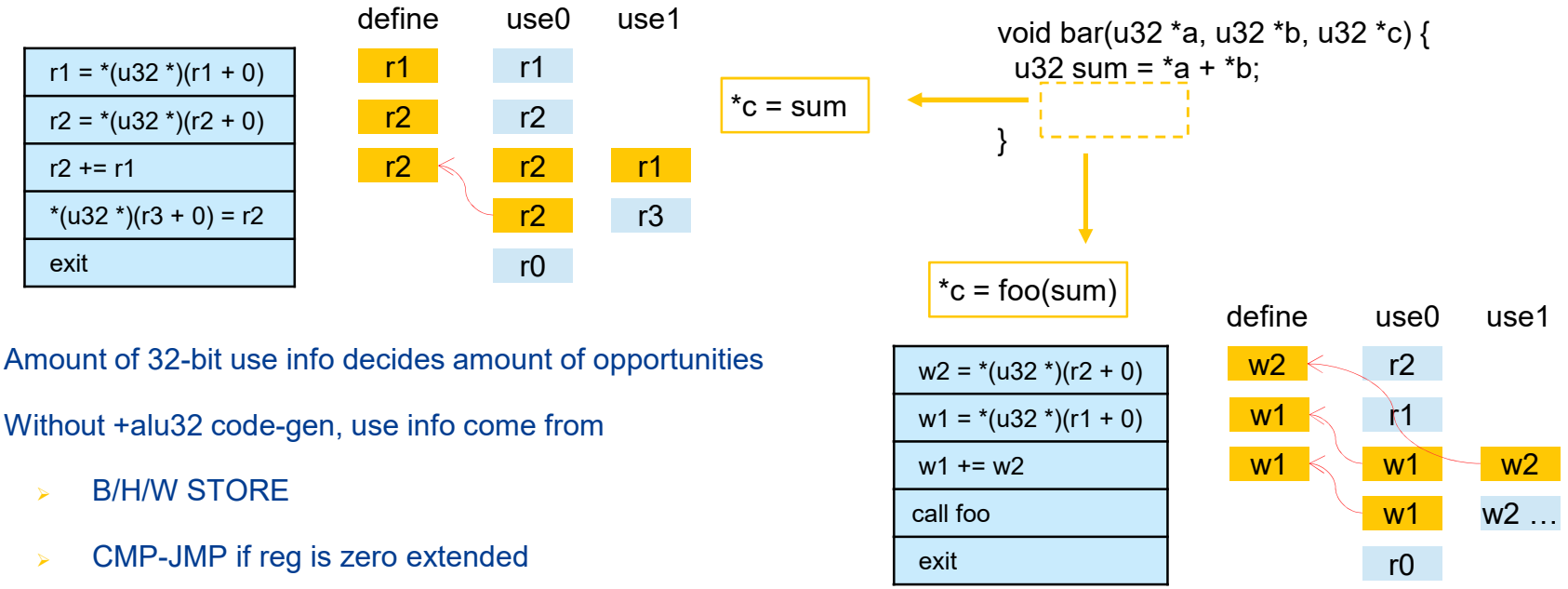
Solution B – Standalone DF Analyzer

NETRONOME

- Work with any input sequence, for example pure ALU64 sequence. Can't leverage LLVM's work
- Optimistic
  - Assume all ALU instructions are **32-bit safe initially**
  - Once find one 64-bit register use, pollute all instructions contributed to the value of this register
- Pessimistic
  - Assume all ALU instructions are **64-bit initially**
  - Mark one ALU insn as 32-bit safe only when all the use of its definition are 32-bit

Solution B – Standalone DF Analyzer

NETRONOME

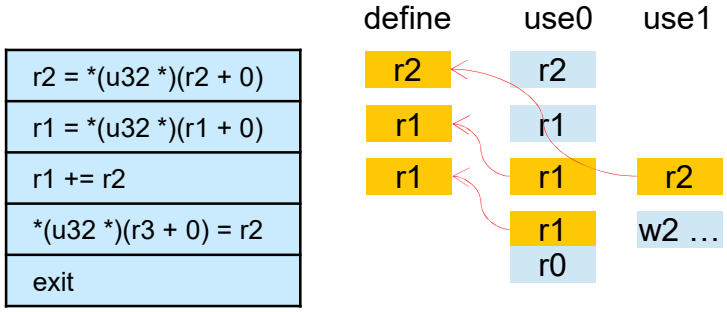


- Amount of 32-bit use info decides amount of opportunities
- Without +alu32 code-gen, use info come from
  - B/H/W STORE
  - CMP-JMP if reg is zero extended
- With +alu32, all ALU32 insn could also produce use info, and analysis could finished quicker

Solution B – Standalone DF Analyzer

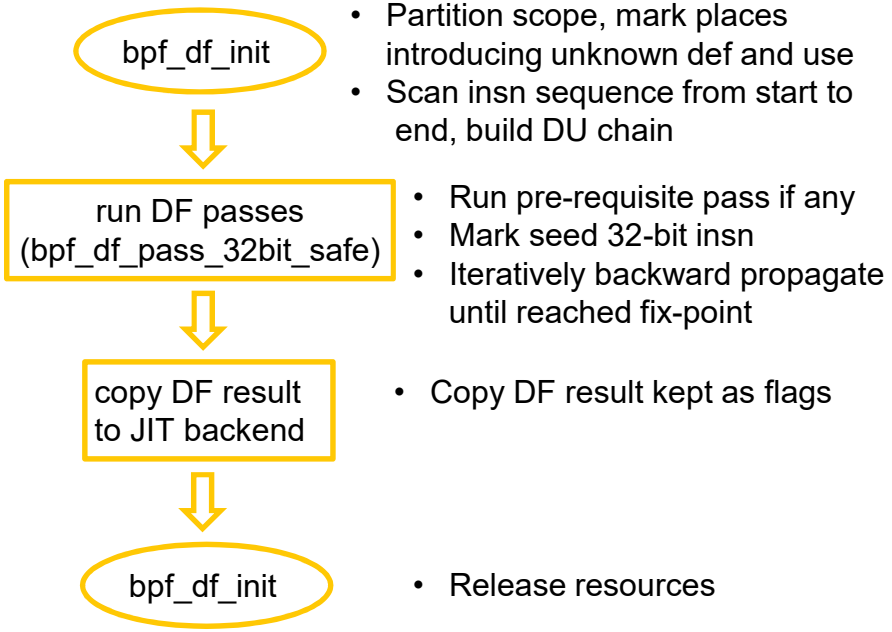
NETRONOME

- A stand-alone, drop-and-work implementation organized as a kernel lib




```
struct bpf_du_insn {
  struct bpf_du_chain *chain[2];
  unsigned int flags;
};
```

```
struct bpf_du_chain {
  struct bpf_du_insn *insn;
  struct bpf_du_chain *next;
  unsigned int flags;
};
```



- Implementation available as RFC
- Could be enhanced with previous CFG infrastructure

  https://lwn.net/Articles/753724 to become a global DF analyzer

Solution B – Standalone DF Analyzer

NETRONOME

- Algorithm pseudo code for local analyzer

```
ext_defs[MAX_BPF_REG] = false;
// insn define the current value of R
act_defs[MAX_BPF_REG] = NULL;

bpf_df_init:
  for_each_insn_in_the_sequence
    if (cur_insn == jmp or call)
      dst_insn.flag |= FLAG_IS_JMP_DST

  for_each_insn_in_the_sequence
    if (cur_insn.flag & FLAG_IS_JMP_DST)
      ext_defs[0..MAX_BPF_REG] = true

    if (cur_insn == call or exit or jmp )
      add_unknown_use(act_def[arg_regs])
      or act_defs[0..MAX_BPF_REG] = NULL

    cur_insn.u2d[0] = act_defs[cur_insn.dst_reg]
    act_defs[cur_insn.dst_reg].d2u.next = cur_insn;
    cur_insn.u2d[1] = act_defs[cur_insn.src_reg]
    act_defs[cur_insn.src_reg].d2u.next = cur_insn;
```

```
bpf_df_pass_32bit_safe:
  for_each_insn_in_the_sequence
    if (cur_insn == ST_B/H/W && has_single_use(cur_insn.u2d[1]))
      cur_insn.u2d[1].flag |= FLAG_IS_32BIT_SAFE
    else if (cur_insn == ALU32) {
      if (has_single_use(cur_insn.u2d[0])
        cur_insn.u2d[0].flag |= FLAG_IS_32BIT_SAFE
      if (BPF_X && has_single_use(cur_insn.u2d[1])
        cur_insn.u2d[1].flag |= FLAG_IS_32BIT_SAFE
    }

  do {
    changed = false
    for_each_insn_in_the_sequence
      changed |= propagate_32bit_safe(cur_insn)
  while (changed)

propagate_32bit_safe:
  changed = false
  if (!(insn.flag & FLAG_IS_32BIT_SAFE))
    return false

  if (insn == alu64 or alu32 except right shift) {
    if (has_single_use(insn.u2d[0])
      insn.u2d[0].flag |= FLAG_IS_32BIT_SAFE
    if (BPF_X && has_single_use(insn.u2d[1])
      insn.u2d[1].flag |= FLAG_IS_32BIT_SAFE
  }
  return changed;
```

Solution B – Standalone DF Analyzer

NETRONOME

- Build global analyser on top of CFG infrastructure
  - Split each register into hi/lo, hi0 ~ lo10, lo0 ~ lo10, do classic live variable analysis
  - Calculate LiveIn and LiveOut for hi registers, meaning whether high 32-bit are live (used later) when entering and leaving one basic block. For 32-bit safety, we care about Out(s) which could tell us the use in successor blocks

$$Out(s) = \cup_{s' \in succ(s)} In(s')$$

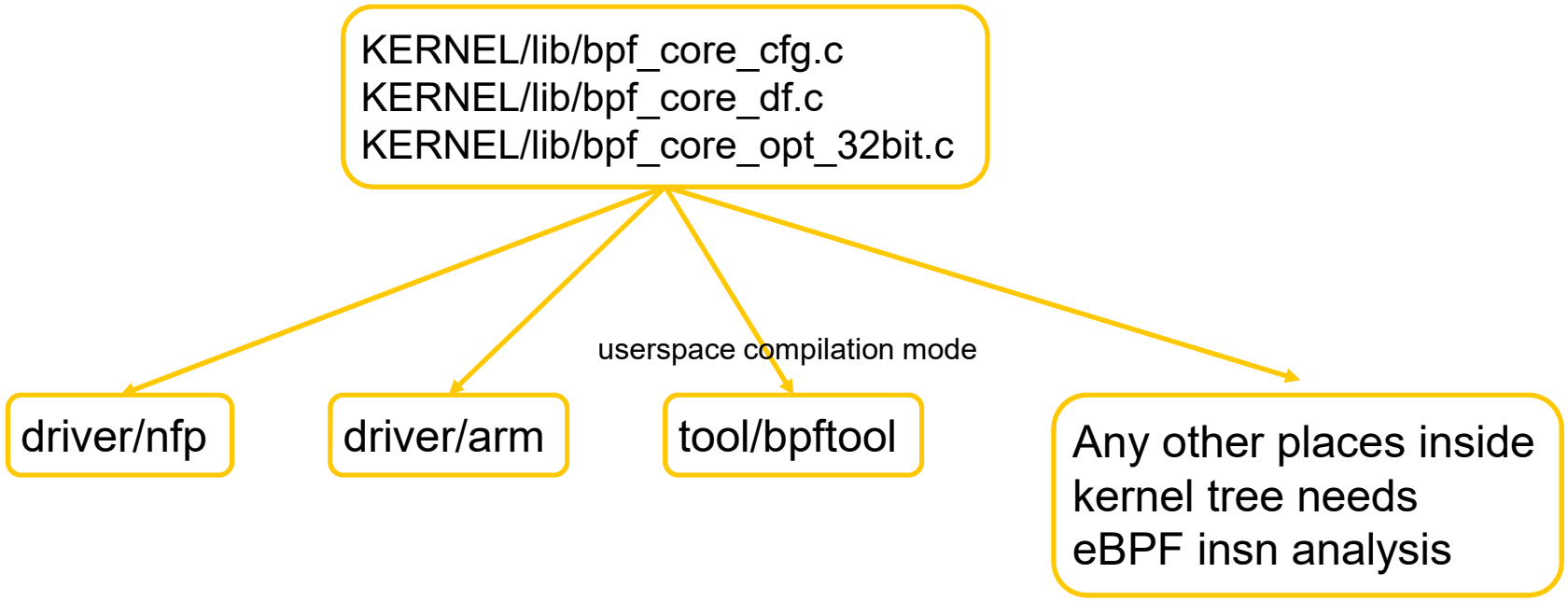
$$In(s) = Gen(s) \cup (Out(s) - Kill(s))$$

  - Gen(s) function is complex than classic Gen(s) as we need insn DU chain to propagate from def to use. Can't simply finish initialize Gen(s) set using insn scan

Solution B – Standalone DF Analyzer

NETRONOME

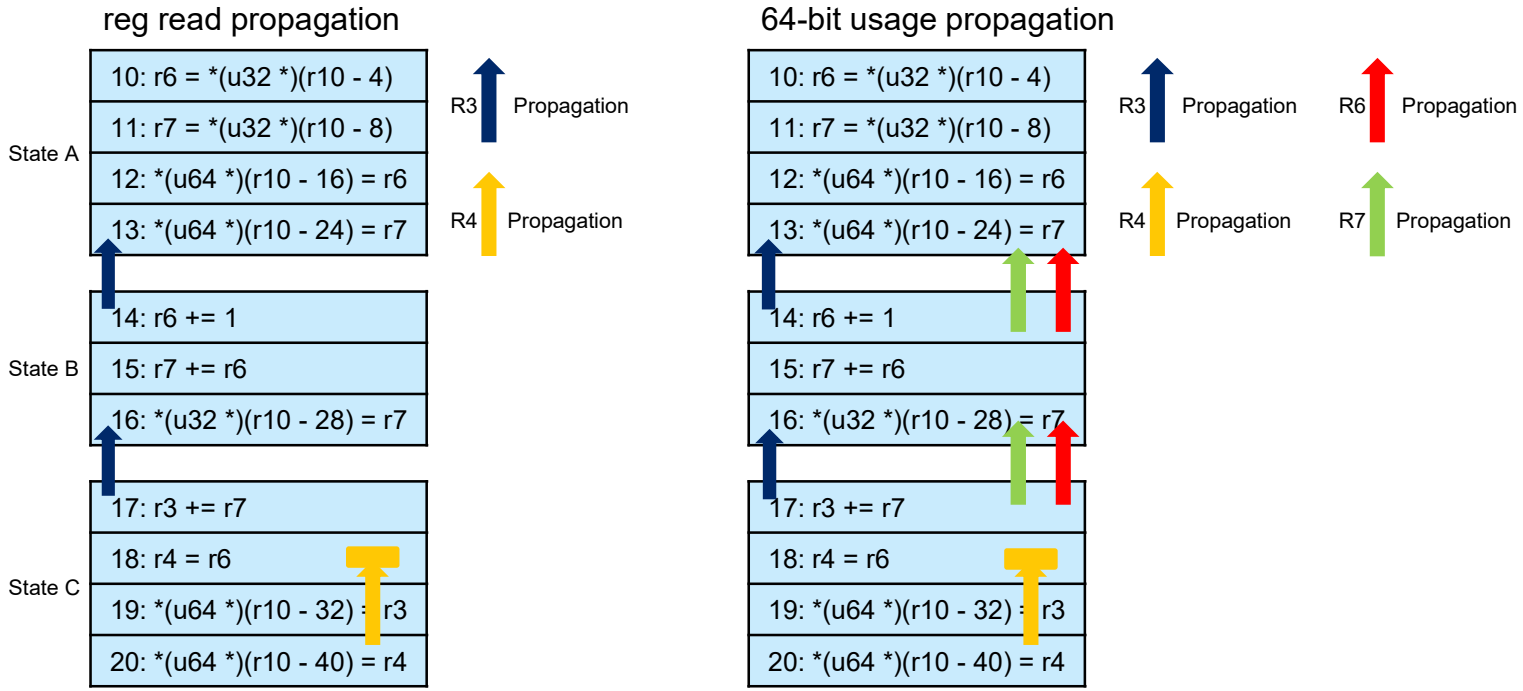
Ideally should compile for both userspace and kernel space



Solution C - Enhance Verifier DF analyzer

NETRONOME

- Verifier has a light “DF analyzer”, designed for releasing more path prune opportunities
- It is integrated with path walker, collects and processes information while walking insn
- Fit scenarios which do not need memorize historical information, but doesn’t fit well otherwise



Solution C - Enhance Verifier DF analyzer

NETRONOME

- 32-bit analysis algorithm based on verifier DF analyzer
  - On solution B, one insn is 32-bit safe if its definition has 32-bit use only, this requires def<->use dual link
  - Verifier DF analyzer is based on code path walking, we don't know whether all uses has been visited, so can only build use->def singular chain
  - Therefore, the algorithm is using 64-bit "polluting". Initially all instructions are considered as 32-bit safe, then whenever there is one 64-bit use of one definition, that insn is polluted as 64-bit
  - 64-bit polluting could propagate from definition to use. For example, A = B + C, definition A has 64-bit use, insn A is polluted, uses B and C must also be 64-bit as they are forming A, therefore definition insn of B and C should be polluted as well

r7 += r6
r3 += r7
*(u64 *)(r10 - 28) = r3

Need to pollute insns that define r3, r6 and r7 and any one further backward iteratively.

```
insn_mark_stack[stack_idx++] = insn_idx;
while (stack_idx) {
  def_idx = insn_mark_stack[--stack_idx];
  aux[def_idx].full_ref = true;

  u2d0 = aux[def_idx].u2d[0];
  if (u2d0 >= 0 && !aux[u2d0].full_ref)
  insn_mark_stack[stack_idx++] = u2d0;

  u2d1 = aux[def_idx].u2d[1];
  if (u2d1 >= 0 && !aux[u2d1].full_ref)
  insn_mark_stack[stack_idx++] = u2d1;
}
```

Solution C - Enhance Verifier DF analyzer

NETRONOME

- 64-bit read in pruned path need to be propagated upward to parent verifier state
- But 64-bit read propagation is more complex, it won't be screened off by a simple write and such propagation needs to be done iteratively on all relevant registers

```
10: r6 = *(u32 *)(r10 - 4)
11: r4+=r3
12: *(u64 *)(r10 - 16) = r6
13: *(u64 *)(r10 - 24) = r7

14: r4 = 1
14: r6 += r4
16: r5 += r6

17: r4 += r5
18: r5 = 1
19: r4 += r5
20: *(u64 *)(r10 - 40) = r4
```

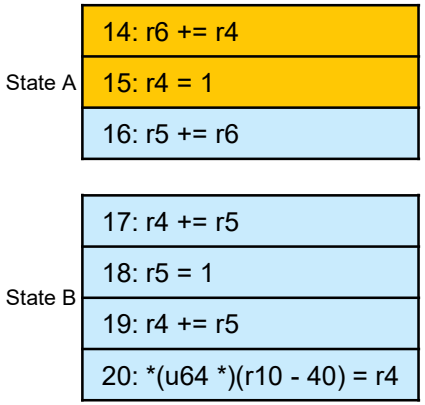
	normal reg read propagation	64-bit reg read propagation
when propagation start	when walking the insn	when walking the insn, but could start later when one relevant reg identified as 64-bit
need historical info	no	yes need to record all insns relevant to one reg
screen off	any write to the reg	write to the reg, but source shouldn't be overlapping with the dest
affect other regs	no	yes

instructions involved generating r4, registers in these insns are all relevant

Solution C - Enhance Verifier DF analyzer

NETRONOME

- Historical information
  - Insns involved with value generation of the reg
  - Because reg state is changing alone with insn walking, so should keep the state when walking the insn
  - For example, r4 in insn 14 contributed to the final value of r4 used in insn 20 which is 64-bit, so it should be propagated to parent state. Insn 15 however screened off r14, if we don't record the reg state when walking insn 14, we will miss this propagation




```
struct bpf_insn_aux_data {
  …
  struct bpf_reg_state_lite reg_state_lite[MAX_BPF_REG];
  struct bpf_verifier_state *parent_vstate;
  s16 u2d[2];
}

struct bpf_reg_state_lite {
  struct bpf_reg_state *parent;
  u32 live;
 };
```

insn walker could visit one insn recursively (disallowed now), or repeatedly, both would break the chain!

Solution C - Enhance Verifier DF analyzer

NETRONOME

- Algorithm pseudo code

```
enum reg_arg_type {
  SRC_OP_0,
  SRC_OP64_0,
  SRC_OP_1,
  SRC_OP64_1,
  SRC_OP64_IMP,
  DST_OP,
  U_DST_OP,
  DST_OP_NO_MARK,
  U_DST_OP_NO_MARK
};

check_reg_arg across verifier need to pass finer arg_type
depending on the position of the src and whether it is 64bit

  check_reg_arg(env, insn->src_reg, SRC_OP|64_0, insn_idx);
  check_reg_arg(env, insn->dst_reg, SRC_OP|64_1, insn_idx);

  dst_type = no overlap between dst_reg and any src ?
             U_DST_OP_NO_MARK : DST_OP_NO_MARK;
  check_reg_arg(env, insn->dst_reg, dst_type, insn_idx);


link_reg_to_def(rstate, type, insn_idx):
  slot_idx = 0 or 1 depending on type
  def_idx = rstate->def_insn_idx;
  insn_aux_data[insn_idx].u2d[slot_idx] = def_idx;
```

```
check_reg_arg(reg, type, insn_idx):
  rstate = cur_frame->regs + insn_idx
  if (type == SRC_*) {
    link_use_to_def(rstate, type, insn_idx)
    mark_reg_read(… type == SRC_*64_*);
  } else {
    rstate->live |= REG_LIVE_WRITTEN;
    if (t == U_DST_OP || t == U_DST_OP_NO_MARK)
      rstate->live |= REG_LIVE_WRITTEN_UNIQUE;

    rstate->def_insn_idx = insn_idx;
    if (insn_idx != INVALID_INSN_IDX)
      copy_reg_state_lite(insn_aux[insn_idx].regs_lite, regs);
  }
```

```
mark_reg_read:
  …
  if (!64bit_read)
    return;
  return mark_reg_read64(…)
```

```
mark_reg_read64:
  while (parent) {
    if (writes &&
        state->live &
        REG_LIVE_WRITTEN_UNIQUE)
      break;
    parent->live |= REG_LIVE_READ64;
    state = parent;
    parent = state->parent;
    writes = true;
  }
  if (no_mark_insn)
    return;
  return mark_insn_64bit(def_idx);
```

```
mark_insn_64bit:
 insn_mark_stack[stack_idx++] = insn_idx;
 while (stack_idx) {
   def_idx = insn_mark_stack[--stack_idx];
   aux[def_idx].full_ref = true;

   u2d0 = aux[def_idx].u2d[0];
   if (u2d0 >= 0 && !aux[u2d0].full_ref) {
     insn_mark_stack[stack_idx++] = u2d0;
     mark_reg_read64(aux[def_idx].u2d0.dst_reg,
                     no_mark_insn)
   }

   u2d1 = aux[def_idx].u2d[1];
   if (u2d1 >= 0 && !aux[u2d1].full_ref) {
     insn_mark_stack[stack_idx++] = u2d1;
     mark_reg_read64(aux[def_idx].u2d1.dst_reg,
                     no_mark_insn)
   }
 }
```

Which approach to go?

NETRONOME

- Verifier
  - Better to focus on verification, offer reliable and fast verification, no bothering of optimization
  - DF analysis based on dynamic insn walking requires recording historical information
  - Path prune however make it very difficult to collect such information on such scope
- Classic CFG + DF analysis
  - Reliable and has sophisticated algorithms
  - Redo LLVM's work, heavy for kernel space
- Reuse information passed down from LLVM through 32-bit subreg ISA
  - Throw the heavy lift work to user space static compiler who is also really good at
  - Lack of some instructions (JMP32 etc) is causing trouble
  - **Best to follow this approach and enhance eBPF ISA ?**

Thank you!