Towards Faster Trace Filters using eBPF and JIT

Suchakrapani Datt Sharma

Dec 11, 2014

École Polytechnique de Montréal

Laboratoire **DORSAL**

Agenda

Recap

- Research Updates

Investigations

- What's the status of BPF?
- Benefits of eBPF & JIT in tracing
- eBPF with kernel tracing
- Early experiments & results

What's Next

- Modify experiments!
- Investigate bytecode generation techniques

Recap

Research Focus : Integrated and streamlined framework for tracing & debugging, dynamic instrumentation

Extensions

- Investigate the use of JIT compilation in tracing and debugging context
- Explore how efficient bytecode generation and JITing can be achieved

Investigations

As of now,

- Tracing is fast, but its components are isolated
- Complex filters and scripts can be expensive

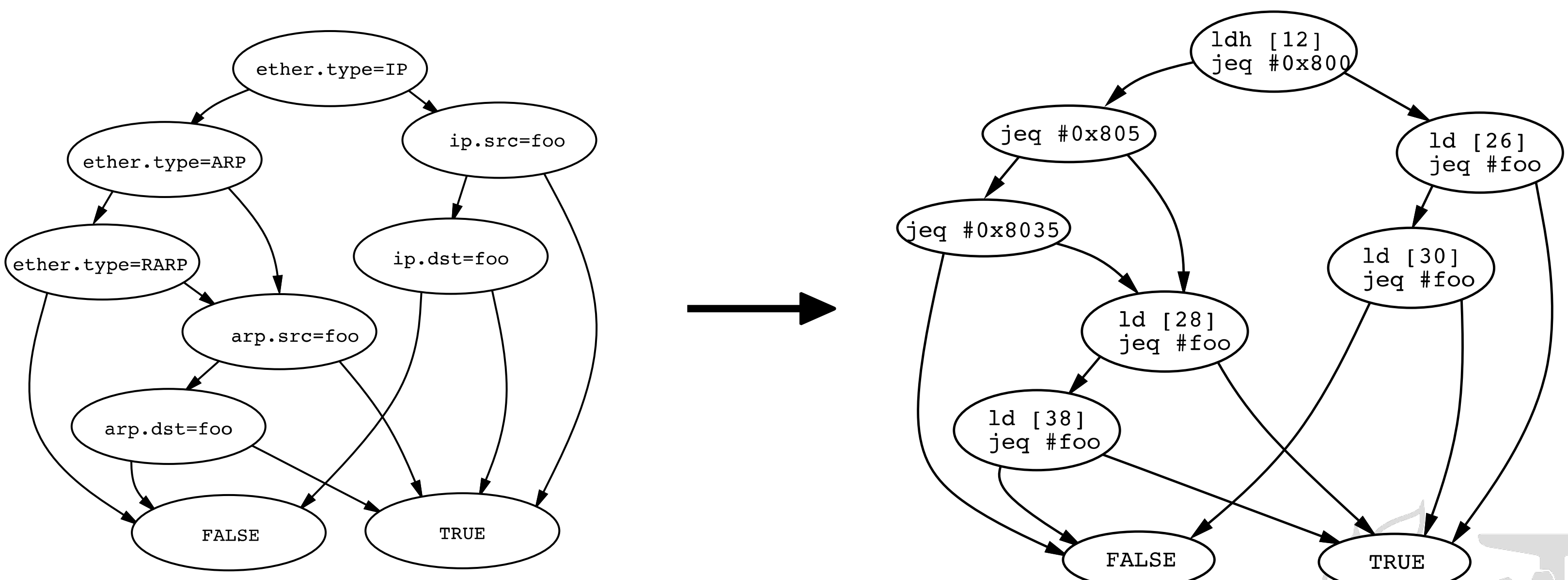
What can be done?

- Uniform framework for trace filters/scripts
  - Extensible but with low overhead
- Improve underlying techniques.
  - JIT when necessary/available [2]
  - Optimized bytecode and JIT [2, 3, 5]

Investigations

Berkeley Packet Filter (BPF)

- Filter expressions → Bytecode → Interpret
- Fast, small, in-kernel packet & syscall filtering [6]
- Register based, switch-dispatch interpreter



Investigations

Berkeley Packet Filter (BPF)

- Filter expressions → Bytecode → Interpret
- Fast, small, in-kernel packet & syscall filtering [6]
- Register based, switch-dispatch interpreter

Current Status of BPF

- Extension for trace filtering (ftrace)
- BPF+JIT for filtering [1, 6]
- Evolved to *extended* BPF (eBPF) [1, 6]
  - BPF maps, *bpf* syscall
  - More registers (64 bit), back jumps, safety

Investigations

Why eBPF in Tracing

- Primarily, for filters & script driven tracing
- Expressions → Bytecode → **JIT**

  ↳ **Interpret**
- Add bulky features to tracing, at low cost
  - Fast stateful kernel event filtering?
- Ktap's Dtrace-*ish* approach but not heavyweight
- A more uniform way of filtering events

Investigations

Initial Experiments (Kernel)

- Custom module with a custom probe for **`netif_receive_skb`** and **`sched_switch`** events

```
// tick

IF ((device_name == "lo") AND (protocol == IP) AND (length > 100))
{
    TRACEPOINT();
}

// tock
```

- Apply simple eBPF, eBPF+JIT, hardcoded filter
- Measure $t_{filter} + t_{tracepoint}$ in probe handler
- Observe code generated by eBPF JIT vs hardcoded filter

Investigations

Short Simple Filter

Hardcoded :

```
if ((dev->name[0] == “l”) && (dev->name[1] == “o”))
{
    trace_netif_receive_skb_filtered(skb);
}
```

Compare “l”

```
42:     cmpb   $0x6c,(%r12)
47:     je     b8
:
:
b8:     cmpb   $0x6f,0x1(%r12)
be:     jne    49                  ; FLASE
```

Compare “o”

Investigations

Short Simple Filter

eBPF Bytecode :

R2 = ctx

R3 = *(dev->name)
R4 = 0x6f6c

```
static struct bpf_insn insn_prog[] = {
    BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_1, 0),
    BPF_LDX_MEM(BPF_DW, BPF_REG_3, BPF_REG_2, 0), /* ctx->arg1 */
    BPF_LDX_MEM(BPF_DW, BPF_REG_4, BPF_REG_1, 8), /* ctx->arg2 */
    BPF_JMP_REG(BPF_JEQ, BPF_REG_3, BPF_REG_4, 3), /* compare arg1 & arg2 */
    BPF_LD_IMM64(BPF_REG_0, 0), /* FALSE */
    BPF_EXIT_INSN(),
    BPF_LD_IMM64(BPF_REG_0, 1), /* TRUE */
    BPF_EXIT_INSN(),
};
```

Sample modules with some more eBPF filters :

- https://gist.github.com/tuxology/68fbd813b6eb84fb9766
- https://gist.github.com/tuxology/1d00223dfa4b93c1031b

Investigations

Short Simple Filter

eBPF JITed :

```
 0:    push   %rbp
 1:    mov    %rsp,%rbp
 4:    sub    $0x228,%rsp
 b:    mov    %rbx,-0x228(%rbp)
12:    mov    %r13,-0x220(%rbp)
       mov    %r14,-0x218(%rbp)
       mov    %r15,-0x210(%rbp)
27:    xor    %eax,%eax
29:    xor    %r13,%r13
2c:    mov    0x0(%rdi),%rsi
       mov    0x0(%rsi),%rdx
       mov    0x8(%rdi),%rcx
38:    cmp    %rcx,%rdx
```

```
3b:    je     0x0000000000000049
3d:    movabs $0x0,%rax        ;FALSE
47:    jmp    0x0000000000000053
49:    movabs $0x1,%rax        ;TRUE
53:    mov    -0x228(%rbp),%rbx
       mov    -0x220(%rbp),%r13
       mov    -0x218(%rbp),%r14
68:    mov    -0x210(%rbp),%r15
6f:    leaveq
70:    retq
```

Make some space on stack

Save callee saved regs

Clear A and X

Compare R3, R4

Load ctx args to R3 and R4

Jump to **TRUE**

Restore regs

One-to-one JITing. More opportunity is in improving bytecode generation

Investigations

Some more filters

netif_receive_skb_filter

Same as before but a bit longer

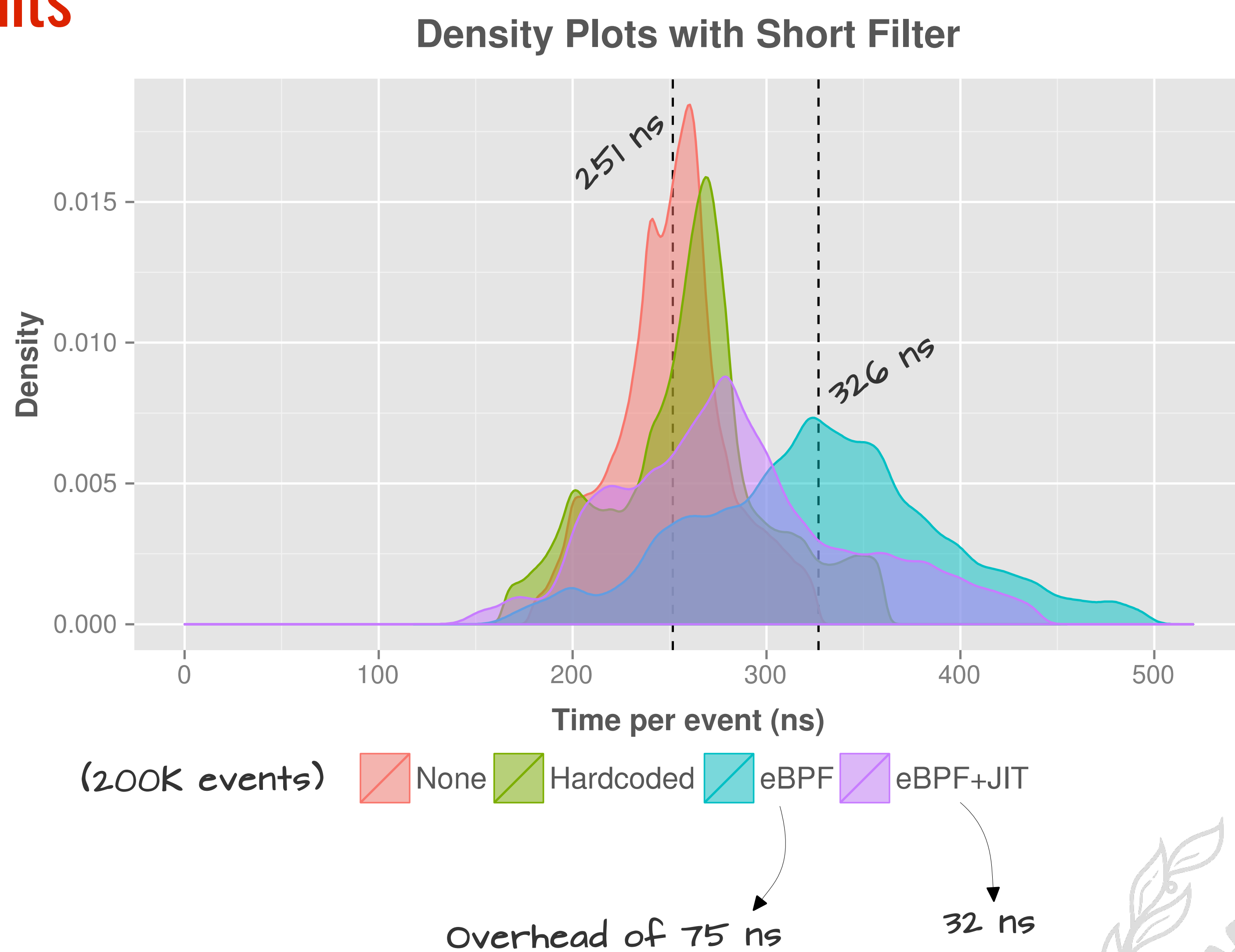
```
if ((dev->name[0] == “l”) && (dev->name[1] == “o”) &&
   (skb->protocol == 8) && (skb->len > 100))
{
     trace_netif_receive_skb_filter(skb);
}
```

sched_switch_filter

```
if ((memcmp(prev->comm, comm, 4) == 0) && (prev->state == 0)
{
     trace_sched_switch_filter(skb);
}
```

Investigations

Results



Investigations

Results

What's Next

Inferences

- Trace filtering with JIT is visibly better
- So, is it any good?
  - Based on feedback, need to revise experiments
  - Not a complete picture yet, remove irregularities

Going Further

- Complex filters, have a better test framework
- Explore specialization and generation of eBPF bytecode
- Put everything in userspace for tighter control

References

[1] https://kernel.googlesource.com/pub/scm/linux/kernel/git/ast/bpf/

[2] Run-Time Bytecode Specialization, Masuhara H., Yonezawa A., *PADO '01 Proceedings of the Second Symposium on Programs as Data Objects,* ACM (*2001)*

[4] Optimizing Lua using run-time type specialization, Schröder M, *B. Thesis (2012)*

[5] Virtual-Machine Abstraction and Optimization Techniques, Brunthaler S. *Electronic Notes in Theoretical Computer Science 253 (2009)*

[6] https://www.kernel.org/doc/Documentation/networking/filter.txt

Questions?

suchakrapani.sharma@polymtl.ca

suchakra on #lttng

Towards Faster Trace Filters using eBPF and JIT

Suchakrapani Datt Sharma

Dec 11, 2014

École Polytechnique de Montréal
Laboratoire **DORSAL**

Agenda

Recap

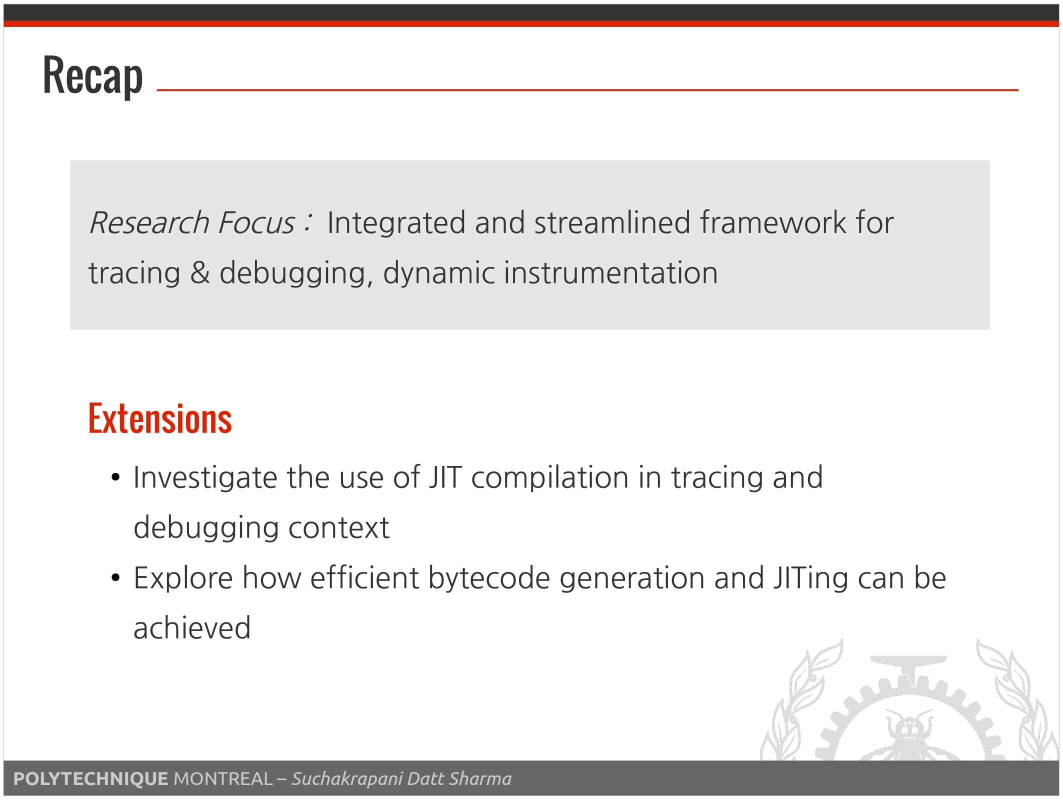
- Research Updates

Investigations

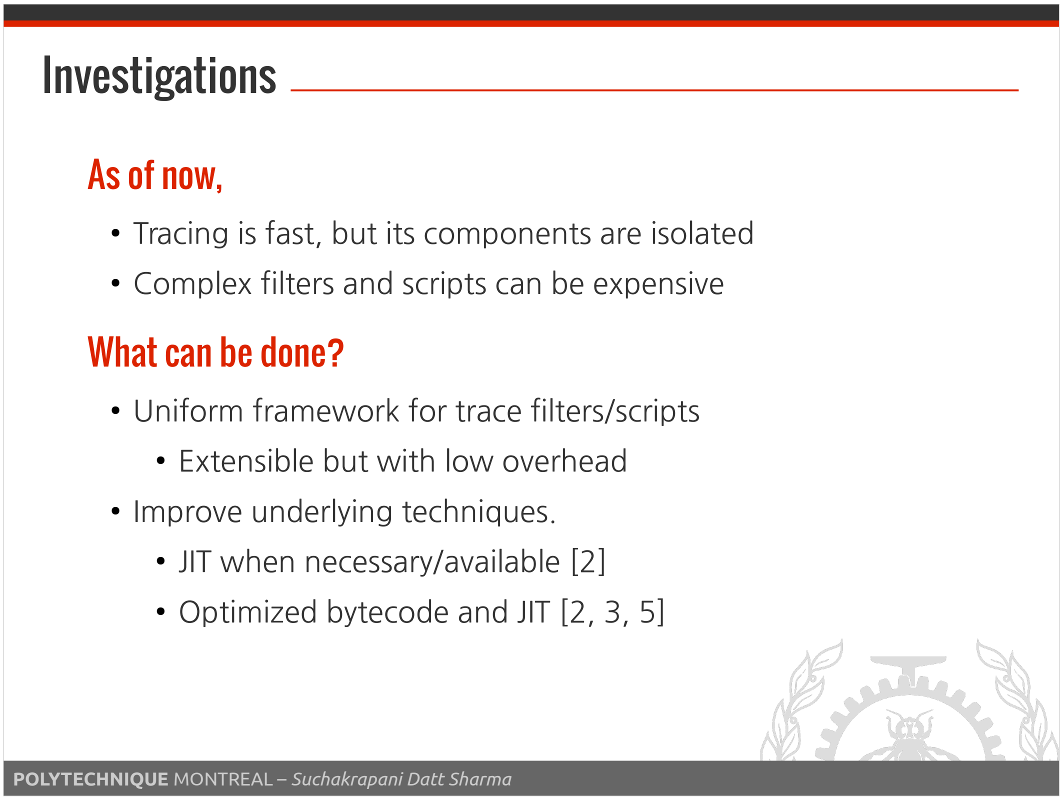
- What's the status of BPF?
- Benefits of eBPF & JIT in tracing
- eBPF with kernel tracing
- Early experiments & results

What's Next

- Modify experiments!
- Investigate bytecode generation techniques



- JIT has been there for quite long and has been recently been used for trace filtering as well
- Need to make bytecode generation as well as JITing efficient



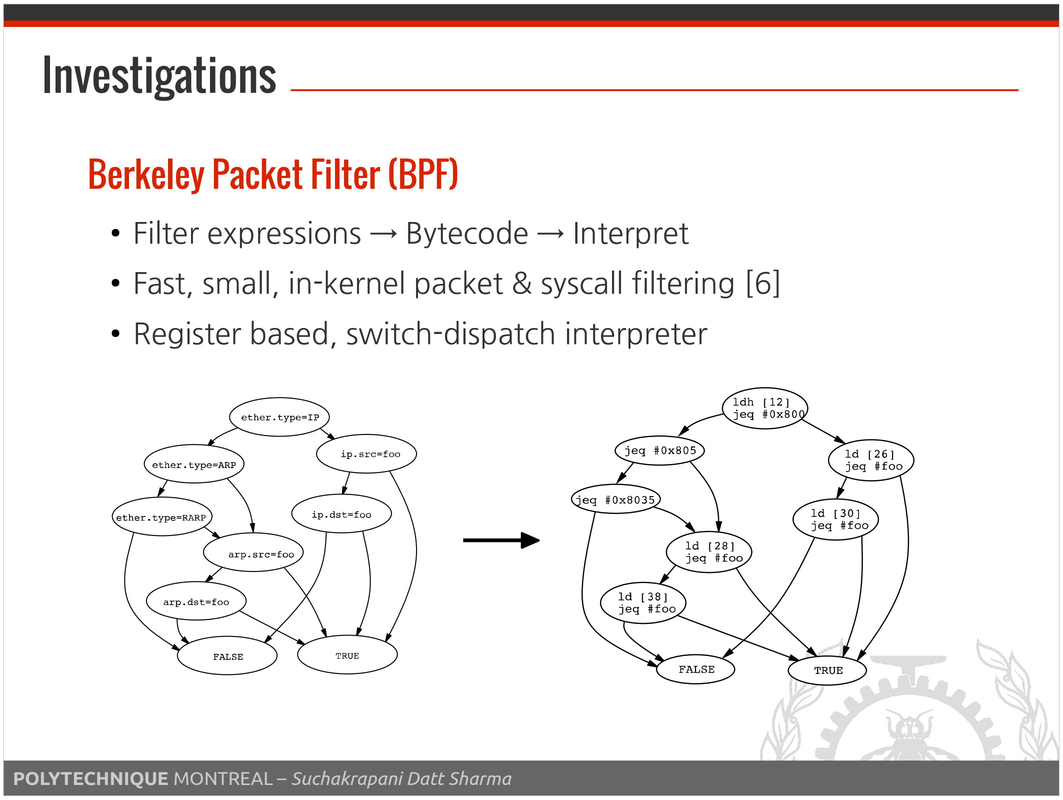
- With latest techniques and work of pioneers, we have achieved very high tracing speeds and minimum overhead – well and good

- But adding more features, newer techniques will drag down the desired performance of tracers

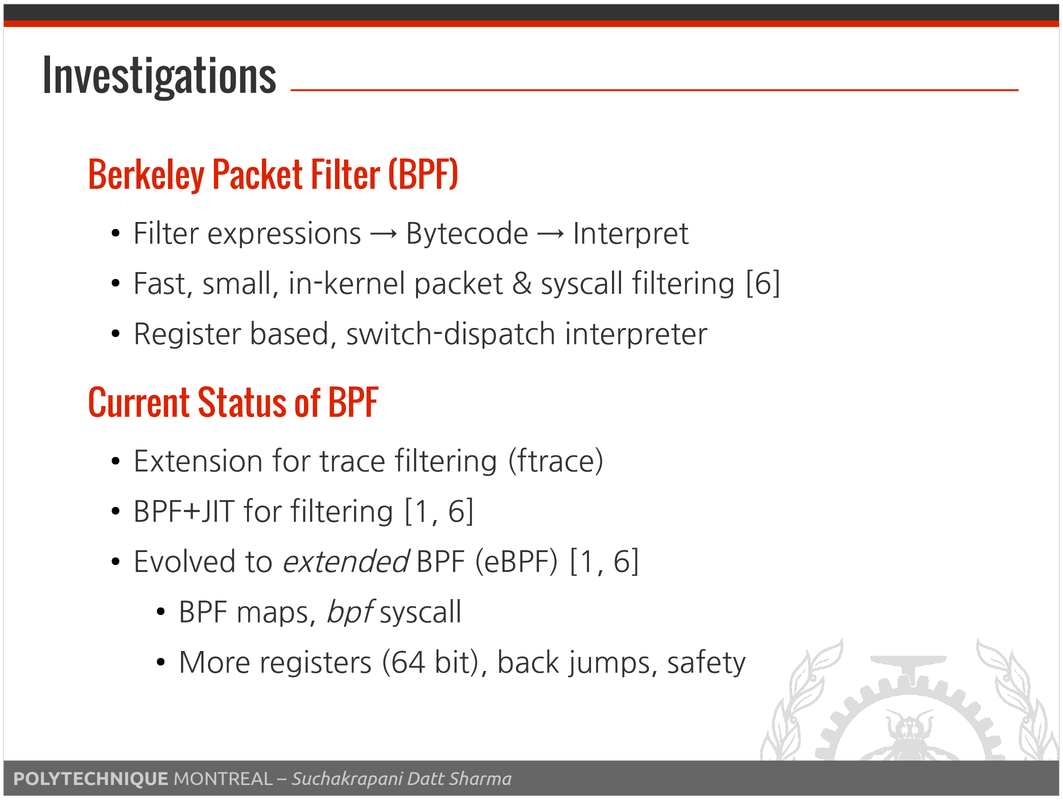
- My goal is to **attack those underlying techniques and algorithms so that tracers become future and feature ready and have uniformity**

  - JIT really improvesJIT only when necessary – method or trace

  - Explore opportunities for optimizing – like specializing bytecode or improve JITing techniques
    - Like determine instruction type, using specialized instructions. Similar to LuaJIT

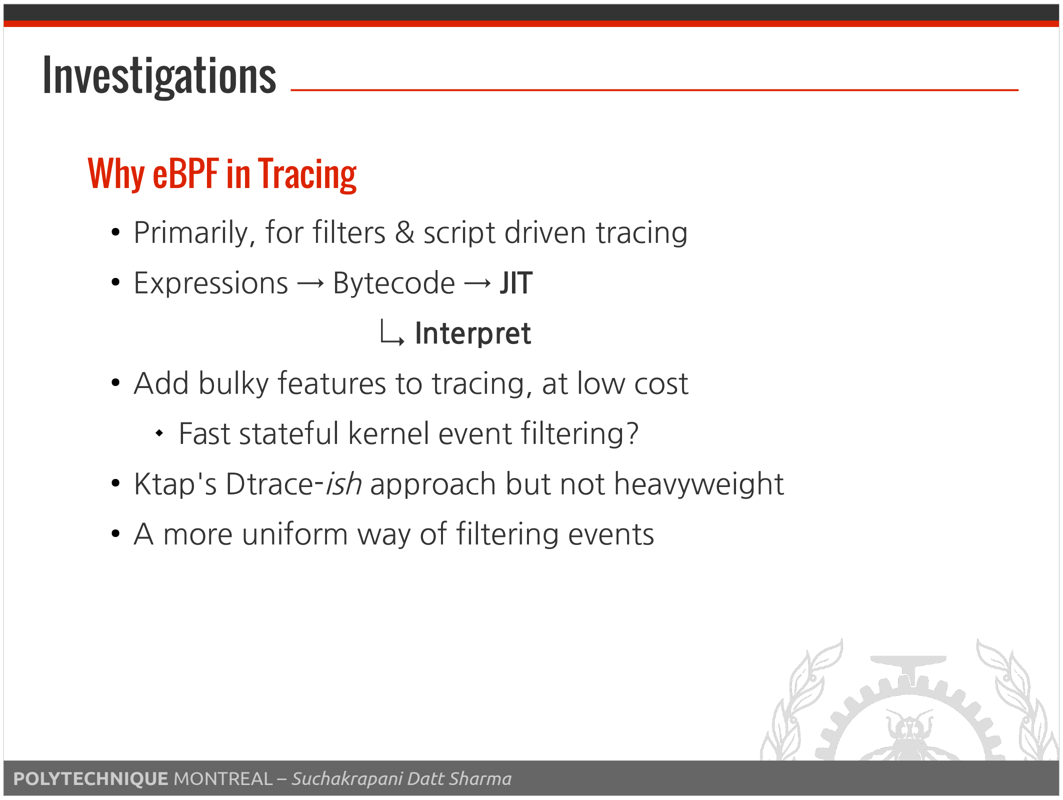


- BPF was simple, two, 32-bit registers
- Rudimentary operations and checking
- Initially designed for packet filtering and replaced the predicate-tree walker



- Extended to 10 64-bit registers with extensions to instructions, better mapping with newer architectures for JITing, better spillage control
- Userspace compilation of bytecode with LLVM/GCC backend, safety checks!
- Its has better acceptance chances to be in kernel – maybe not for tracing use so soon!

- Take care to not blow it to a full VM and adapt it for our use cases



- If we make the infrastructure cheap, we can afford to do bulky things like maintain in-kernel states to enhance filters
  - Get me all the events that are causing some daemon to be pre-empted very often

- Ktap has tried before to do this to make script based tracing like dtrace with scripts generating bytecode to be interpreted by ktapvm (in kernel)

- EBPF on other hand is an extension of an already existing infra, re-factored, enhanced and can be used anywhere.
  - Libpcap still uses either bpf(kernel – interpreted/jited) or bpf userspace as fallback

Investigations

Initial Experiments (Kernel)

- Custom module with a custom probe for **`netif_receive_skb`** and **`sched_switch`** events

```
// tick

IF ((device_name == "lo") AND (protocol == IP) AND (length > 100))
{
    TRACEPOINT();
}

// tock
```

- Apply simple eBPF, eBPF+JIT, hardcoded filter
- Measure $t_{filter} + t_{tracepoint}$ in probe handler
- Observe code generated by eBPF JIT vs hardcoded filter

Investigations

Short Simple Filter

Hardcoded :

```
if ((dev->name[0] == “l”) && (dev->name[1] == “o”))
{
    trace_netif_receive_skb_filtered(skb);
}
```

Compare “l”

```
42:      cmpb   $0x6c,(%r12)
47:      je     b8
:
:
b8:      cmpb   $0x6f,0x1(%r12)
be:      jne    49                ; FLASE
```

Compare “o”

Investigations

Short Simple Filter

eBPF Bytecode :

R2 = ctx

R3 = *(dev->name)
R4 = 0x6f6c

```
static struct bpf_insn insn_prog[] = {
    BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_1, 0),
    BPF_LDX_MEM(BPF_DW, BPF_REG_3, BPF_REG_2, 0), /* ctx->arg1 */
    BPF_LDX_MEM(BPF_DW, BPF_REG_4, BPF_REG_1, 8), /* ctx->arg2 */
    BPF_JMP_REG(BPF_JEQ, BPF_REG_3, BPF_REG_4, 3), /* compare arg1 & arg2 */
    BPF_LD_IMM64(BPF_REG_0, 0), /* FALSE */
    BPF_EXIT_INSN(),
    BPF_LD_IMM64(BPF_REG_0, 1), /* TRUE */
    BPF_EXIT_INSN(),
};
```

Sample modules with some more eBPF filters :

- https://gist.github.com/tuxology/68fbd813b6eb84fb9766
- https://gist.github.com/tuxology/1d00223dfa4b93c1031b

Investigations

Short Simple Filter

eBPF JITed :

```
 0:    push   %rbp
 1:    mov    %rsp,%rbp
 4:    sub    $0x228,%rsp
 b:    mov    %rbx,-0x228(%rbp)
12:    mov    %r13,-0x220(%rbp)
19:    mov    %r14,-0x218(%rbp)
       mov    %r15,-0x210(%rbp)
27:    xor    %eax,%eax
29:    xor    %r13,%r13
2c:    mov    0x0(%rdi),%rsi
       mov    0x0(%rsi),%rdx
34:    mov    0x8(%rdi),%rcx
38:    cmp    %rcx,%rdx
```

```
3b:    je     0x0000000000000049
3d:    movabs $0x0,%rax        ;FALSE
47:    jmp    0x0000000000000053
49:    movabs $0x1,%rax        ;TRUE
53:    mov    -0x228(%rbp),%rbx
       mov    -0x220(%rbp),%r13
61:    mov    -0x218(%rbp),%r14
68:    mov    -0x210(%rbp),%r15
6f:    leaveq
70:    retq
```


One-to-one JITing. More opportunity is in improving bytecode generation

Investigations

Some more filters

netif_receive_skb_filter

Same as before but a bit longer

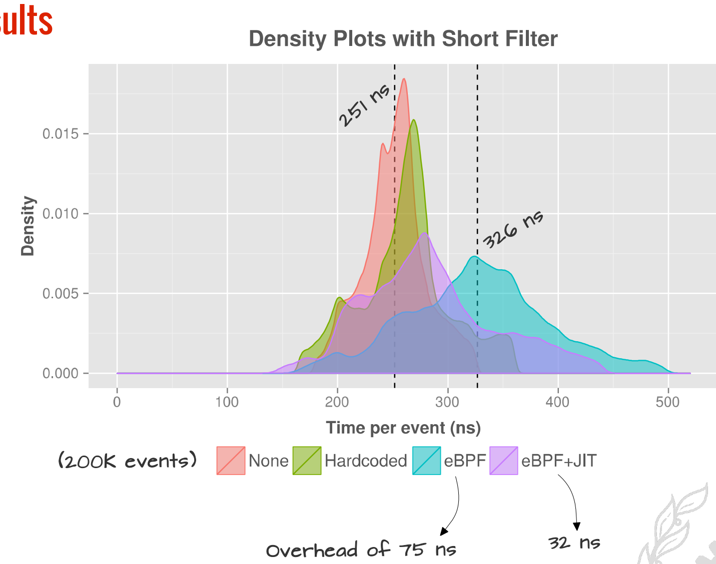
```
if ((dev->name[0] == “l”) && (dev->name[1] == “o”) &&
   (skb->protocol == 8) && (skb->len > 100))
{
    trace_netif_receive_skb_filter(skb);
}
```

sched_switch_filter

```
if ((memcmp(prev->comm, comm, 4) == 0) && (prev->state == 0)
{
    trace_sched_switch_filter(skb);
}
```

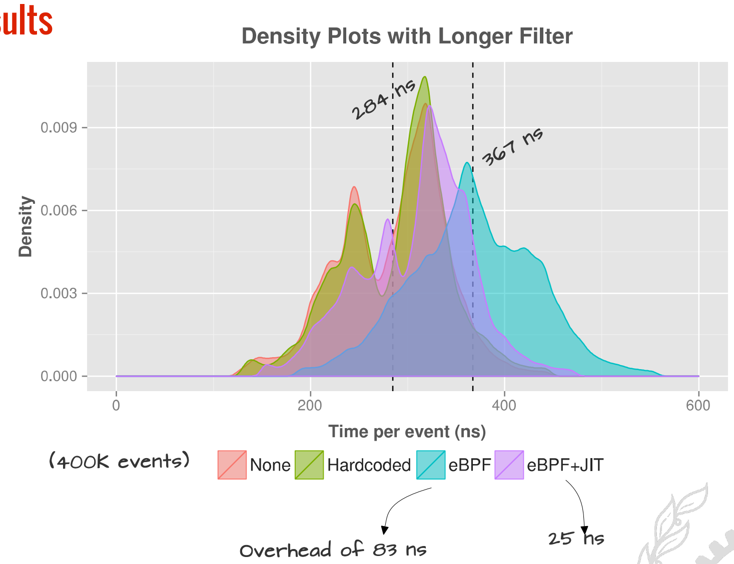
Investigations

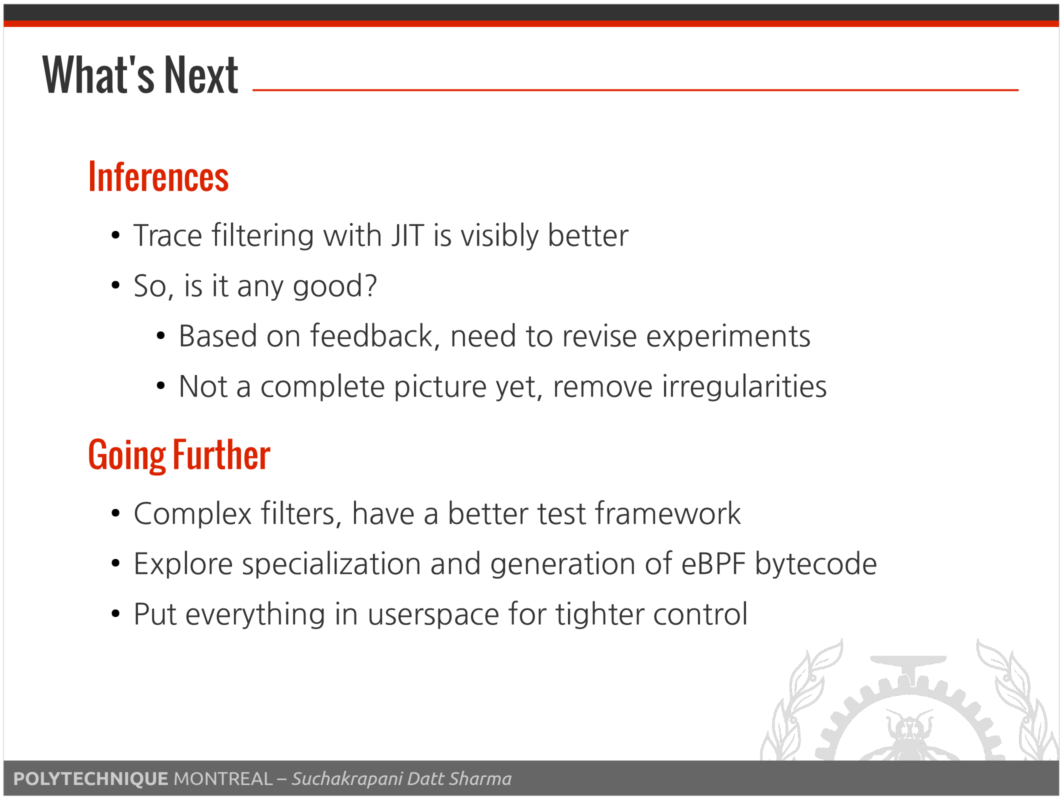
Results



Investigations

Results





- All PASS / All FAIL filters
- Time saved in typical trace record scenarios because of filtering

References

[1] https://kernel.googlesource.com/pub/scm/linux/kernel/git/ast/bpf/

[2] Run-Time Bytecode Specialization, Masuhara H., Yonezawa A., *PADO '01 Proceedings of the Second Symposium on Programs as Data Objects,* ACM (*2001)*

[4] Optimizing Lua using run-time type specialization, Schröder M, *B. Thesis (2012)*

[5] Virtual-Machine Abstraction and Optimization Techniques, Brunthaler S. *Electronic Notes in Theoretical Computer Science 253 (2009)*

[6] https://www.kernel.org/doc/Documentation/networking/filter.txt

Questions?

suchakrapani.sharma@polymtl.ca

suchakra on #lttng