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]
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, \texttt{bpf} syscall
  - More registers (64 bit), back jumps, safety
Why eBPF in Tracing

- Primarily, for filters & script driven tracing
- Expressions $\rightarrow$ Bytecode $\rightarrow$ JIT
  $\downarrow$ Interpret
- Add bulky features to tracing, at low cost
  - Fast stateful kernel event filtering?
- Ktap's Dtrace-\textit{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:**

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

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

Compare “l”

Compare “o”
Short Simple Filter

eBPF Bytecode :

```c
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 3b: je 0x0000000000000049
1: mov %rsp,%rbp 3d: movabs $0x0,%rax ;FALSE
4: sub $0x228,%rsp 47: jmp 0x0000000000000053
b: mov %rbx,-0x228(%rbp) 49: movabs $0x1,%rax ;TRUE
12: mov %r13,-0x220(%rbp) 53: mov -0x228(%rbp),%rbx
13: mov %r14,-0x218(%rbp) 5a: mov -0x220(%rbp),%r13
mov %r15,-0x210(%rbp) 61: mov -0x218(%rbp),%r14
27: xor %eax,%eax 68: mov -0x210(%rbp),%r15
29: xor %r13,%r13 6f: leaveq
30: xor %r13,%r13 70: retq
31: mov 0x0(%rdi),%rsi
32: 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
5a: 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.
Some more filters

**netif_receive_skb_filter**

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

**sched_switch_filter**

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

Results

Density Plots with Short Filter

Time per event (ns)

Density

(200K events)

Overhead of 75 ns

326 ns

251 ns

32 ns

None

Hardcoded

eBPF

eBPF+JIT
Investigations

Results

Density Plots with Longer Filter

(400K events)

None Hardcoded eBPF eBPF+JIT

Overhead of 83 ns

25 ns
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


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
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

- 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
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]

• 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 improves JIT 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
Investigations

Berkeley Packet Filter (BPF)

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

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

- 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
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

- 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

```c
// 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
**Short Simple Filter**

Hardcoded:

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

`cmpb $0x6c,(%r12)`  
`je b8`

`cmpb $0x6f,0x1(%r12)`  
`jne 49 ; FLASE`

- Compare 'l'
- Compare 'l'
- Compare 'o'
- Compare 'o'
Short Simple Filter

eBPF Bytecode:

static struct bpf_insn insn_prog[] = {
    BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_1, 0), /* ctx->arg1 */
    BPF_LDX_MEM(BPF_DW, BPF_REG_3, BPF_REG_2, 0), /* 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:

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

netif_receive_skb_filter

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

sched_switch_filter

```c
if ((memcmp(prev->comm, comm, 4) == 0) && (prev->state == 0)
{
    trace_sched_switch_filter(skb);
}
```
Density Plots with Short Filter

Overhead of 75 ns

32 ns

(200K events)
Investigations

Results

Density Plots with Longer Filter

- 264 ns
- 261 ns

Time per event (ns)

Density

(400K events)

None  Hardcoded  eBPF  eBPF+JIT

Overhead of 89 ns  25 ns
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

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


Questions?

suchakrapani.sharma@polymtl.ca
suchakra on #lttng