# BPF in LLVM and kernel

2015 aug 19

## $ git diff kernel/bpf/

| classic BPF | extended BPF |
|---|---|
| 2 registers + stack<br>32-bit registers<br>4-byte load/store to stack<br>1-4 byte load from packet<br>Conditional jump forward<br>+, -,  *, … instructions | 10 registers + stack<br>64-bit registers with 32-bit sub-registers<br>1-8 byte load/store to stack, maps, context<br>Same + store to packet<br>Conditional jump forward and backward<br>Same + signed_shift + endian<br>Call instruction<br>tail_call<br>map lookup/update/delete helpers<br>packet rewrite, csum, clone_redirect<br>sk_buff read/write<br>tunnel metadata read/write<br>vlan push/pop<br>hash/array/prog/perf_event map types |

## LLVM backend BPF

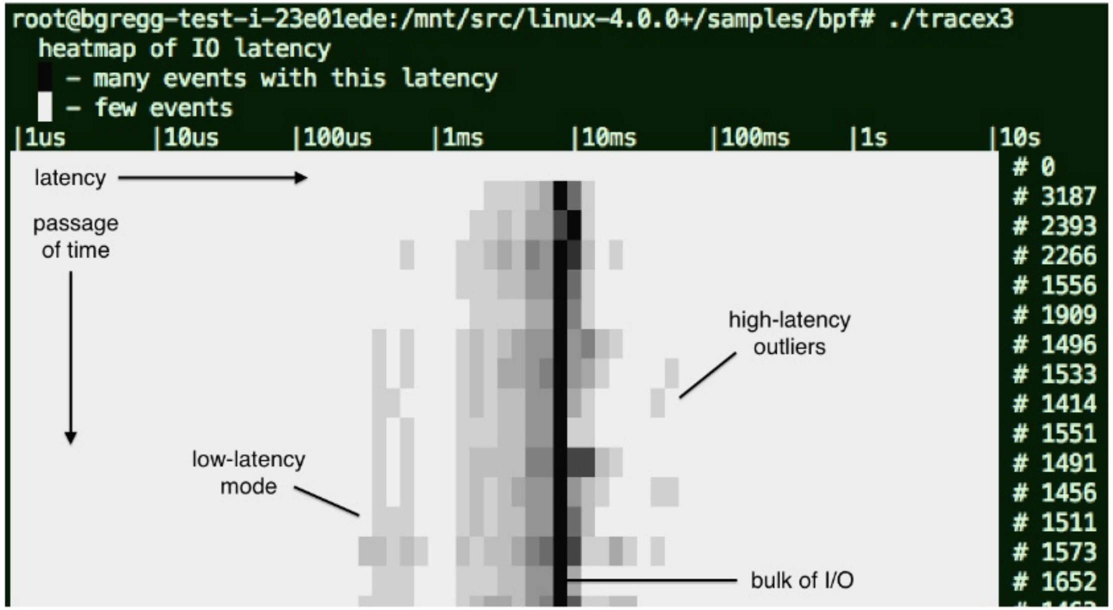
- simple backend, can be used as an example to write new backends
- BPF backend is in LLVM tree since Feb 2015
- will be released as part of 3.7 in August 21, 2015
- clang –O2 –target bpf –c file.c –o file.o

## LLVM backend BPF

- integrated assembler generates ELF
- supports JIT mode (in-memory .c to in-memory bpf binary)
- source & docs
  - https://github.com/llvm-mirror/llvm/tree/master/lib/Target/BPF/
  - http://llvm.org/docs/CodeGenerator.html#the-extended-berkeley-packet-filter-ebpf-backend
- tbd
  - 32-bit sub-registers
  - debug info and builtin_bpf_typeid
  - p4 front-end

# BPF in tracing

- programs can be attached to any kprobe event
- read any kernel data structure
- aggregate into bpf maps
- report to user space



```
root@bgregg-test-i-23e01ede:/mnt/src/linux-4.0.0+/samples/bpf# ./tracex3
  heatmap of IO latency
  █ - many events with this latency
  █ - few events
|1us        |10us       |100us      |1ms        |10ms       |100ms      |1s         |10s
                                                                                     # 0
                                                                                     # 3187
                                                                                     # 2393
                                                                                     # 2266
                                                                                     # 1556
                                                                                     # 1909
                                                                                     # 1496
                                                                                     # 1533
                                                                                     # 1414
                                                                                     # 1551
                                                                                     # 1491
                                                                                     # 1456
                                                                                     # 1511
                                                                                     # 1573
                                                                                     # 1652
```

https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/samples/bpf/tracex3_kern.c

## bpf + perf

- perf record --event file.[co] <command>
  - by Wang Nan @huawei
  - http://thread.gmane.org/gmane.linux.kernel/1975092
  - enables 'perf record' to filter events using eBPF programs
  - existing 'perf report' will visualize collected data
  - on the fly .c compilation or elf load .o
  - generic tools/lib/bpf/libbpf.a

## Writing a BPF Program - Easy Mode

- Write your BPF program in C... inline or in a separate file
- Write a python script that loads and interacts with your BPF program
  - Attach to kprobes, socket, tc filter/action
  - Read/update maps
  - Configuration, complex calculation/correlations
- Iterate on above and re-try...in seconds
- https://github.com/iovisor/bcc

## Hello World!, BPF

```python
from bpf import BPF
from subprocess import call
prog = """
int hello(void *ctx) {
  bpf_trace_printk("Hello, World!\\n");
  return 0;
};
"""
b = BPF(text=prog)
fn = b.load_func("hello", BPF.KPROBE)
BPF.attach_kprobe(fn, "sys_clone")
try:
  call(["cat", "/sys/kernel/debug/tracing/trace_pipe"])
except KeyboardInterrupt:
  pass
```

```
[root@localhost examples]# ./hello_world.py
 python-20662 [001] d..1 1138.551706: : Hello, World!
   tmux-1012  [002] d..1 1139.227627: : Hello, World!
   tmux-1012  [002] d..1 1139.229636: : Hello, World!
  byobu-20664 [006] d..1 1139.235396: : Hello, World!
  byobu-20665 [007] d..1 1139.236660: : Hello, World!
  byobu-20665 [007] d..1 1139.246109: : Hello, World!
^C
```

## bcc/tools

- set of performance observation tools by Brendan Gregg
  - https://github.com/iovisor/bcc/tree/master/tools
  - syncsnoop - Trace sync() syscall
  - pidpersec – Shows the number of new processes created per second
  - vfscount - Counts VFS calls (kernel calls beginning with "vfs_")
  - vfsstat - Traces some common VFS calls and prints per-second summaries
  - much more in bcc/examples/

# task_switch stats with BPF maps

```python
// foo.py
from bpf import BPF
from time import sleep
b = BPF(src_file="foo.c")
fn = b.load_func("count_sched", BPF.KPROBE)
stats = b.get_table("stats")
BPF.attach_kprobe(fn, "finish_task_switch")
# generate many schedule events
for i in range(0, 100): sleep(0.01)
# iterate over elements in 'stats' table
for k, v in stats.items():
  print("task_switch[%5d->%5d]=%u" %
        (k.prev_pid, k.curr_pid, v.value))
```

```
[root@localhost examples]# ./foo.py
task_switch[    0->10779]=100
task_switch[    0-> 3914]=1
task_switch[    0-> 2379]=4
task_switch[    0->   44]=1
task_switch[10779->    0]=100
task_switch[   37->    0]=1
task_switch[    0-> 3134]=5
^C
```

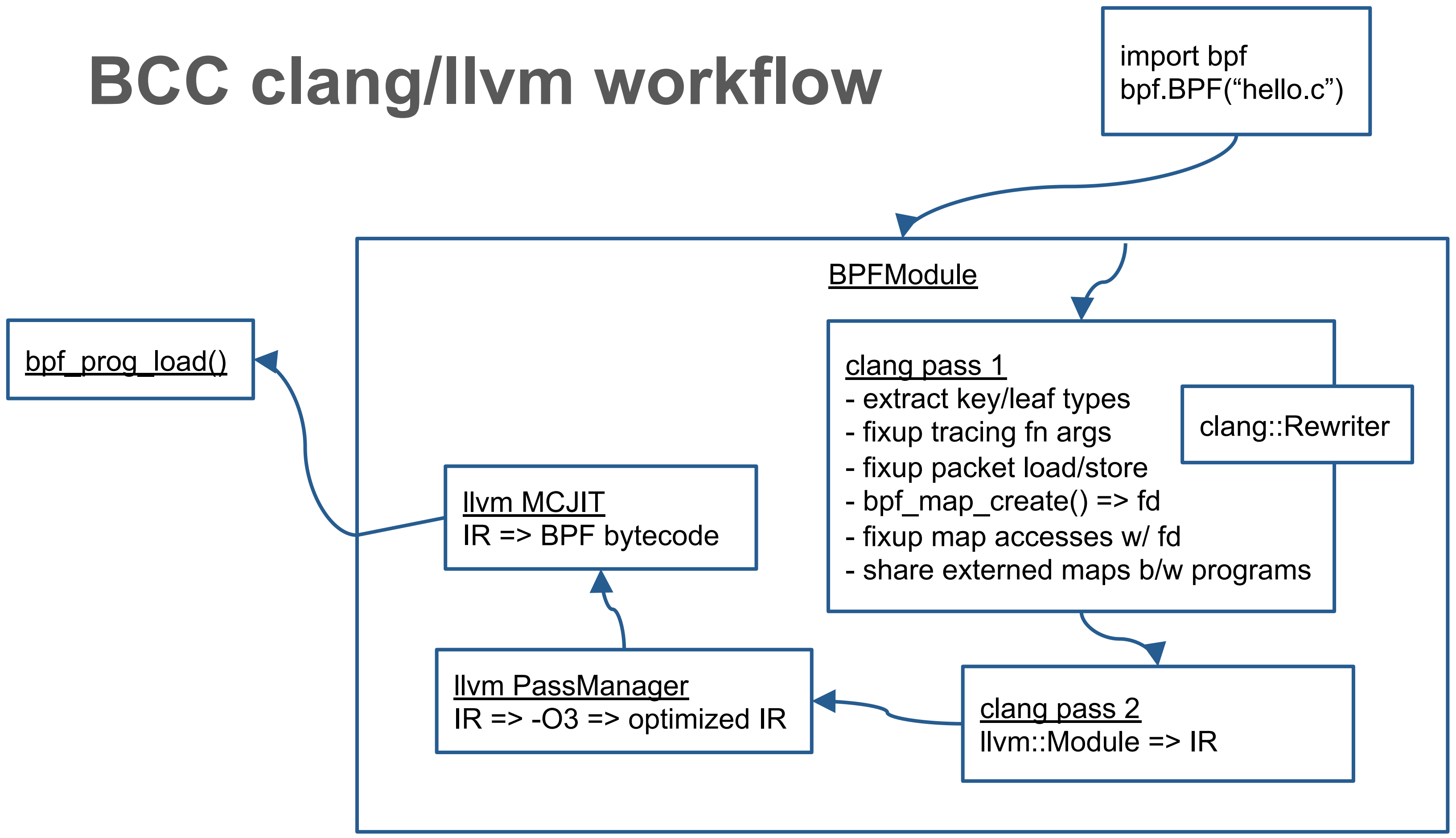
```c
// foo.c
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>
struct key_t {
  u32 prev_pid;
  u32 curr_pid;
};
// map_type, key_type, leaf_type, table_name, num_entry
BPF_TABLE("hash", struct key_t, u64, stats, 1024);
int count_sched(struct pt_regs *ctx, struct task_struct *prev) {
  struct key_t key = {};
  u64 zero = 0, *val;

  key.curr_pid = bpf_get_current_pid_tgid();
  key.prev_pid = prev->pid;
  val = stats.lookup_or_init(&key, &zero);
  (*val)++;
  return 0;
}
```

## Under the Hood

- C API for working with BPF programs - libbpfprog.so
  - JIT compile a C source file to BPF bytecode (using clang+llvm)
  - Load bytecode and maps to kernel with bpf() syscall (no more ELF)
  - Attach 1 or more BPF programs to 1 or more hook points
    - kprobe, socket, tc classifier, tc action
- Python bindings on top of libbpfprog.so - "import bpf"
  - Interactively load/run programs, inspect/update tables
  - Integrate with pyroute2 for attaching to TC

# BCC clang/llvm workflow



## BCC clang::Rewriter

```
#include <uapi/linux/ptrace.h>

int do_request(struct pt_regs *ctx, int req)
{
    bpf_trace_printk("req ptr: 0x%x\n", req);
    return 0;
}
```

```
#include <uapi/linux/ptrace.h>

__attribute__((section(".bpf.fn.do_request")))
int do_request(struct pt_regs *ctx, int req)
{
    ({
       char _fmt[] = "req ptr: 0x%x\n";
       bpf_trace_printk_(_fmt, sizeof(_fmt), ((u64)ctx->di));
    });
    return 0;
}
```

# BCC clang::Rewriter

```
#include <linux/sched.h>
#include <uapi/linux/ptrace.h>

int count_sched(struct pt_regs *ctx,
                struct task_struct *prev)
{
    pid_t p = prev->pid;
    return p != -1;
}
```

```
#include <linux/sched.h>
#include <uapi/linux/ptrace.h>

__attribute__((section(".bpf.fn.count_sched")))
int count_sched(struct pt_regs *ctx, struct task_struct *prev)
{
    pid_t p = ({
      pid_t _val;
      memset(&_val, 0, sizeof(_val));
      bpf_probe_read(&_val, sizeof(_val),
                     ((u64)ctx->di) + offsetof(struct task_struct, pid));
      _val;
    });
    return p != -1;
}
```

## BCC clang::Rewriter

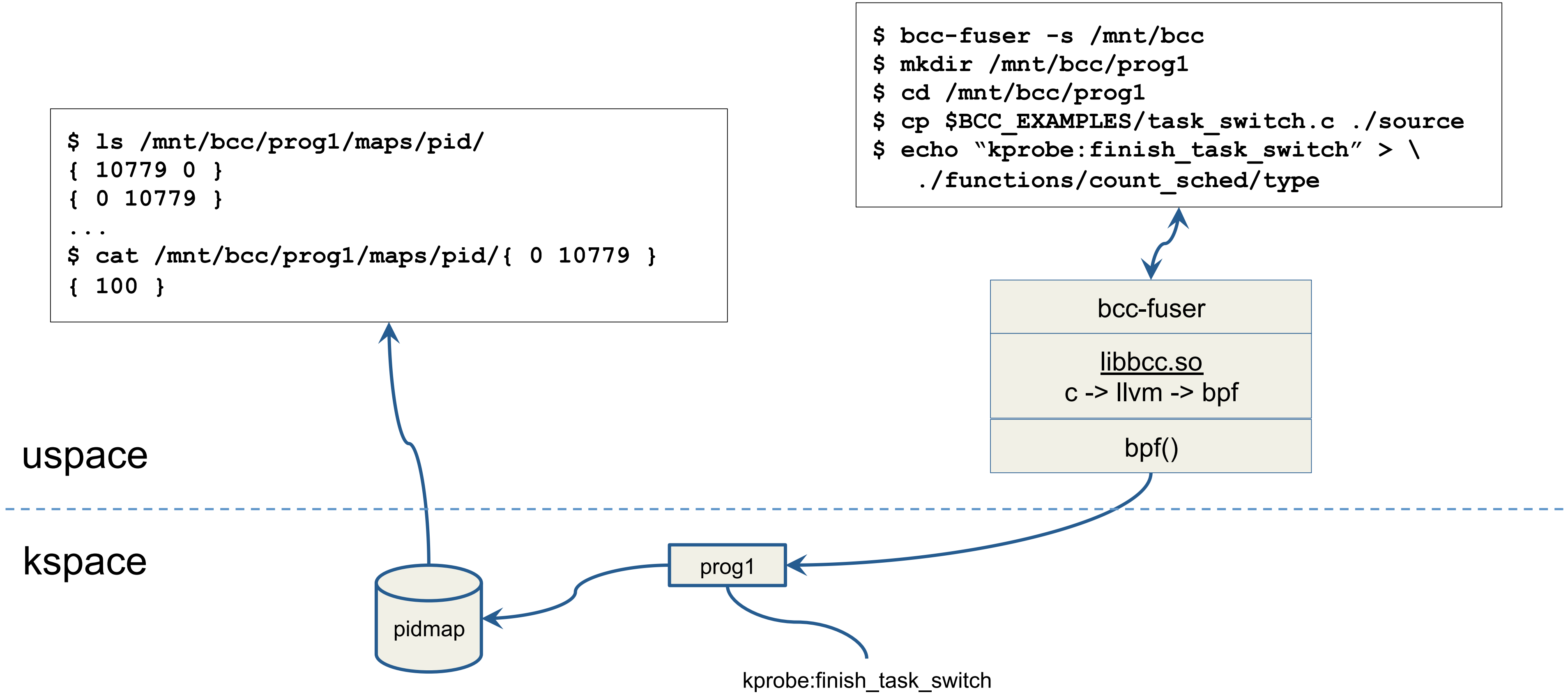
```
#include <uapi/linux/ptrace.h>

int do_request(struct pt_regs *ctx, int req)
{
    bpf_trace_printk("req ptr: 0x%x\n", req);
    return 0;
}
```

```
#include <uapi/linux/ptrace.h>

__attribute__((section(".bpf.fn.do_request")))
int do_request(struct pt_regs *ctx, int req)
{
    ({
       char _fmt[] = "req ptr: 0x%x\n";
       bpf_trace_printk_(_fmt, sizeof(_fmt), ((u64)ctx->di));
    });
    return 0;
}
```

# BPF filesystem (BCC Fuse)



```
$ bcc-fuser -s /mnt/bcc
$ mkdir /mnt/bcc/prog1
$ cd /mnt/bcc/prog1
$ cp $BCC_EXAMPLES/task_switch.c ./source
$ echo "kprobe:finish_task_switch" > \
   ./functions/count_sched/type
```

```
$ ls /mnt/bcc/prog1/maps/pid/
{ 10779 0 }
{ 0 10779 }
...
$ cat /mnt/bcc/prog1/maps/pid/{ 0 10779 }
{ 100 }
```