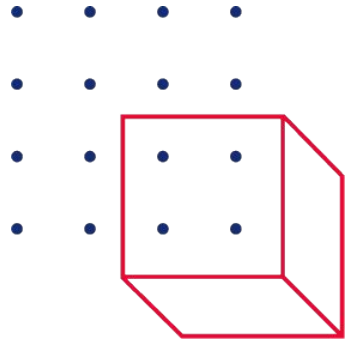
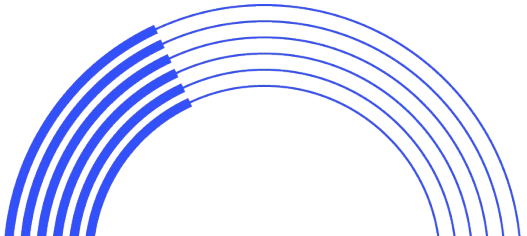
PERCONA LIVE ONLINE MAY 12 - 13th 2021

PingCAP TiDB

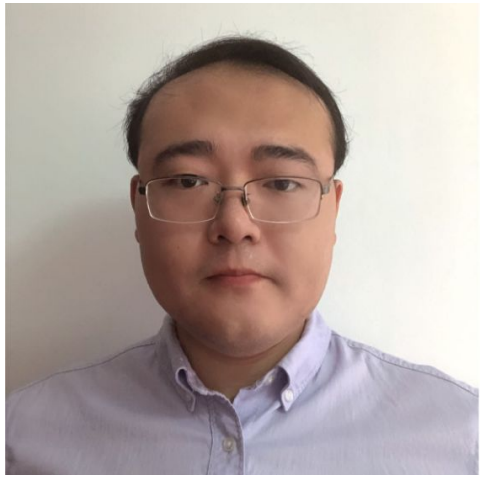


How to develop BPF tools with libbpf + BPF CO-RE

Wenbo Zhang



Speaker



Wenbo Zhang
Engineer at PingCAP

Kernel engineer
Active contributor of libbpf-tools

github@ethercflow

What’s BPF CO-RE?

- BPF portability
- BPF CO-RE (Compile Once – Run Everywhere)

Why we need BPF CO-RE?

- BCC has several major drawbacks

High-level BPF CO-RE mechanics

- Necessary pieces
  - BTF type information
  - compiler (Clang)
  - BPF loader (libbpf)
  - kernel

“Hello world” with libbpf

- minimal.bpf.c

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

char LICENSE[] SEC("license") = "Dual BSD/GPL";

int my_pid = 0;

SEC("tp/syscalls/sys_enter_write")
int handle_tp(void *ctx)
{
        int pid = bpf_get_current_pid_tgid() >> 32;

        if (pid != my_pid)
                return 0;

        bpf_printk("BPF triggered from PID %d.\n", pid);

        return 0;
}
```

- minimal.c

```c
/* Open BPF application */
skel = minimal_bpf__open();
if (!skel) {
        fprintf(stderr, "Failed to open BPF skeleton\n");
        return 1;
}

/* ensure BPF program only handles write() syscalls from our process */
skel->bss->my_pid = getpid();

/* Load & verify BPF programs */
err = minimal_bpf__load(skel);
if (err) {
        fprintf(stderr, "Failed to load and verify BPF skeleton\n");
        goto cleanup;
}

/* Attach tracepoint handler */
err = minimal_bpf__attach(skel);
if (err) {
        fprintf(stderr, "Failed to attach BPF skeleton\n");
        goto cleanup;
}

printf("Successfully started!\n");

for (;;) {
        /* trigger our BPF program */
        fprintf(stderr, ".");
        sleep(1);
}

:
minimal_bpf__destroy(skel);
```

BPF skeleton and BPF app lifecycle

- BPF application typically phases:
  - **Open phase**
  - **Load phase**
  - **Attachment phase**
  - **Tear down phase**

BPF skeleton and BPF app lifecycle

- Generated BPF skeleton has corresponding functions to trigger each phase:
  - *<name>__open()*
  - *<name>__load()*
  - *<name>__attach()*
  - *<name>__destroy()*

```
/* Open BPF application */
skel = minimal_bpf__open();
if (!skel) {
        fprintf(stderr, "Failed to open BPF skeleton\n");
        return 1;
}

/* ensure BPF program only handles write() syscalls from our process */
skel->bss->my_pid = getpid();

/* Load & verify BPF programs */
err = minimal_bpf__load(skel);
if (err) {
        fprintf(stderr, "Failed to load and verify BPF skeleton\n");
        goto cleanup;
}

/* Attach tracepoint handler */
err = minimal_bpf__attach(skel);
if (err) {
        fprintf(stderr, "Failed to attach BPF skeleton\n");
        goto cleanup;
}

printf("Successfully started!\n");

for (;;) {
        /* trigger our BPF program */
        fprintf(stderr, ".");
        sleep(1);
}

:
minimal_bpf__destroy(skel);
```

Combining the open and load phases

- numamove.c#L88

```
obj = numamove_bpf__open_and_load(); Wenbo Zhang, 7 months ago: • li
if (!obj) {
        fprintf(stderr, "failed to open and/or load BPF object\n");
        return 1;
}

err = numamove_bpf__attach(obj);
if (err) {
        fprintf(stderr, "failed to attach BPF programs\n");
        goto cleanup;
}
```

Selective attach

- biolatency.c#L264

```
if (env.queued) {
        obj->links.block_rq_insert =
                bpf_program__attach(obj->progs.block_rq_insert);
        err = libbpf_get_error(obj->links.block_rq_insert);
        if (err) {
                fprintf(stderr, "failed to attach: %s\n", strerror(-err));
                goto cleanup;
        }
}
obj->links.block_rq_issue =
        bpf_program__attach(obj->progs.block_rq_issue);
err = libbpf_get_error(obj->links.block_rq_issue);
if (err) {
        fprintf(stderr, "failed to attach: %s\n", strerror(-err));
        goto cleanup;
}
```

Selective load

- ext4dist.c#L195

```
if (should_fallback()) {
        bpf_program__set_autoload(skel->progs.fentry1, false);
        bpf_program__set_autoload(skel->progs.fentry2, false);
        bpf_program__set_autoload(skel->progs.fentry3, false);
        bpf_program__set_autoload(skel->progs.fentry4, false);
        bpf_program__set_autoload(skel->progs.fexit1, false);
        bpf_program__set_autoload(skel->progs.fexit2, false);
        bpf_program__set_autoload(skel->progs.fexit3, false);
        bpf_program__set_autoload(skel->progs.fexit4, false);
} else {
        bpf_program__set_autoload(skel->progs.kprobe1, false);
        bpf_program__set_autoload(skel->progs.kprobe2, false);
        bpf_program__set_autoload(skel->progs.kprobe3, false);
        bpf_program__set_autoload(skel->progs.kprobe4, false);
        bpf_program__set_autoload(skel->progs.kretprobe1, false);
        bpf_program__set_autoload(skel->progs.kretprobe2, false);
        bpf_program__set_autoload(skel->progs.kretprobe3, false);
        bpf_program__set_autoload(skel->progs.kretprobe4, false);
}
```

Custom load and attach

- runqlen.c#L122

```c
static int open_and_attach_perf_event(int freq, struct bpf_program *prog,
                                      struct bpf_link *links[])
{
        struct perf_event_attr attr = {
                .type = PERF_TYPE_SOFTWARE,
                .freq = 1,
                .sample_period = freq,
                .config = PERF_COUNT_SW_CPU_CLOCK,
        };
        int i, fd;

        for (i = 0; i < nr_cpus; i++) {
                fd = syscall(__NR_perf_event_open, &attr, -1, i, -1, 0);
                if (fd < 0) {
                        /* Ignore CPU that is offline */
                        if (errno == ENODEV)
                                continue;
                        fprintf(stderr, "failed to init perf sampling: %s\n",
                                strerror(errno));
                        return -1;
                }
                links[i] = bpf_program__attach_perf_event(prog, fd);
                if (libbpf_get_error(links[i])) {
                        fprintf(stderr, "failed to attach perf event on cpu: "
```

Multiple handlers for the same event

- hardirqs.bpf.c#L31

```
SEC("tracepoint/irq/irq_handler_entry")
int handle__irq_handler(struct trace_event_raw_irq_handler_entry *ctx)
{
        struct irq_key key = {};
        struct info *info;

        bpf_probe_read_kernel_str(&key.name, sizeof(key.name), ctx->__data);
        info = bpf_map_lookup_or_try_init(&infos, &key, &zero);
        if (!info)
                return 0;
        info->count += 1;
        return 0;
}

SEC("tp_btf/irq_handler_entry")
int BPF_PROG(irq_handler_entry)
{
        u64 ts = bpf_ktime_get_ns();
```

- hardirqs.bpf.c#L31

```
SEC("tp_btf/irq_handler_entry")
int BPF_PROG(irq_handler_entry1, int i
{
        struct irq_key key = {};
        struct info *info;

        bpf_probe_read_kernel_str(&key
        info = bpf_map_lookup_or_try_i
        if (!info)
                return 0;
        info->count += 1;
        return 0;
}

SEC("tp_btf/irq_handler_entry")
int BPF_PROG(irq_handler_entry2)
```

Reading kernel structure's fields

- BCC way:
  pid_t pid = task->pid;
- Libbpf + BPF_PROG_TYPE_TRACING way:
  pid_t pid = task->pid;
- BPF_PROG_TYPE_TRACING + BPF CO-RE way:
  pid_t pid = __builtin_preserve_access_index(({ task->pid; }));
- Non-CO-RE libbpf way:
  pid_t pid; bpf_probe_read(&pid, sizeof(pid), &task->pid);
- CO-RE+libbpf way:
  pid_t pid; bpf_core_read(&pid, sizeof(pid), &task->pid);
  or
  bpf_probe_read(&pid, sizeof(pid), __builtin_preserve_access_index(&task->pid));

eg: bootstrap.bpf.c

How to tell if a field exists？

- bpf_core_field_exists()

  pid_t pid = bpf_core_field_exists(task->pid) ? BPF_CORE_READ(task, pid) : -1;

How to deal with kernel API changes ?

- biolatency.bpf.c#L12

```
extern int LINUX_KERNEL_VERSION __kconfig;

SEC("tp_btf/block_rq_insert")
int block_rq_insert(u64 *ctx)
{
        /**
         * commit a54895fa (v5.11-rc1) changed tracepoint argument list
         * from TP_PROTO(struct request_queue *q, struct request *rq)
         * to TP_PROTO(struct request *rq)
         */
        if (LINUX_KERNEL_VERSION <= KERNEL_VERSION(5, 10, 0)) Roman Sudar
                return trace_rq_start((void *)ctx[1], false);
        else
                return trace_rq_start((void *)ctx[0], false);
}
```

Control BPF behavior with global variable

- cpudist.bpf.c#L46

```
const volatile bool targ_per_process = false;
const volatile bool targ_per_thread = false;
const volatile bool targ_offcpu = false;
const volatile bool targ_ms = false;
const volatile pid_t targ_tgid = -1;

if (targ_tgid != -1 && targ_tgid != tgid)
        return;

tsp = bpf_map_lookup_elem(&start, &pid);
if (!tsp || ts < *tsp)
        return;

if (targ_per_process)
        id = tgid;
else if (targ_per_thread)
        id = pid;
else
        id = -1;
```

```
obj = cpudist_bpf__open();
if (!obj) {
        fprintf(stderr, "failed to open BPF object\n");
        return 1;
}

/* initialize global data (filtering options) */
obj->rodata->targ_per_process = env.per_process;
obj->rodata->targ_per_thread = env.per_thread;
obj->rodata->targ_ms = env.milliseconds;
obj->rodata->targ_offcpu = env.offcpu;
obj->rodata->targ_tgid = env.pid;

pid_max = get_pid_max();
if (pid_max < 0) {
        fprintf(stderr, "failed to get pid_max\n");
        return 1;
}

bpf_map__resize(obj->maps.start, pid_max);
if (!env.per_process && !env.per_thread)
        bpf_map__resize(obj->maps.hists, 1);
else
        bpf_map__resize(obj->maps.hists, pid_max);

err = cpudist_bpf__load(obj);
```

Reduce map pre-allocation overhead

- cpudist.bpf.c#L46

```
struct {
        __uint(type, BPF_MAP_TYPE_HASH);
        __uint(max_entries, MAX_ENTRIES);
        __type(key, struct request *);
        __type(value, u64);
        __uint(map_flags, BPF_F_NO_PREALLOC);
} start SEC(".maps");
```

Determine the map size at runtime

- cpudist.bpf.c#L46

```
struct {
        __uint(type, BPF_MAP_TYPE_HASH);
        __type(key, u32);
        __type(value, struct hist);
} hists SEC(".maps");

bpf_map__resize(obj->maps.start, pid_max);
if (!env.per_process && !env.per_thread)
        bpf_map__resize(obj->maps.hists, 1);
else
        bpf_map__resize(obj->maps.hists, pid_max);

err = cpudist_bpf__load(obj);
```

Use global variables instead of map

- cpudist.bpf.c#L46

```
__u64 counts[NR_SOFTIRQS] = {};
struct hist hists[NR_SOFTIRQS] = {};
static int print_count(struct softirqs_bpf__bss *bss)
{
        const char *units = env.nanoseconds ? "nsecs" : "usecs";
        __u64 count;
        __u32 vec;

        printf("%-16s %6s%5s\n", "SOFTIRQ", "TOTAL_", units);

        for (vec = 0; vec < NR_SOFTIRQS; vec++) {
                count = __atomic_exchange_n(&bss->counts[vec], 0,
                                            __ATOMIC_RELAXED);
                if (count > 0)
                        printf("%-16s %11llu\n", vec_names[vec], count);
        }
```

Learn More

- Articles
  - BPF CO-RE (Compile Once – Run Everywhere)
  - BCC to libbpf conversion guide
  - Building BPF applications with libbpf-bootstrap
  - Tips & tricks for writing libbpf-tools

- Projects
  - libbpf-bootstrap
  - libbpf-tools

- other resources
  - website: https://pingcap.com/
  - Slack: #everyone on Slack
  - Twitter: https://twitter.com/PingCAP

PingCAP
Make Growth Easier.

About Us

PingCAP is a software service provider committed to delivering one-stop enterprise-grade database solutions.

TiDB is an open-source, distributed New SQL database for elastic scale and real-time analytics.

THANK YOU !

PERCONA
LIVEONLINE
MAY 12 - 13th
2021