第三届 eBPF开发者大会

www.ebpftravel.com

深入解析eBPF的性能开销

汇报人：杨月顺
导师：陈莉君教授

性能工程实验室

中国·西安

目录：

第三届 eBPF开发者大会

www.ebpftravel.com

① 解析eBPF性能开销来源

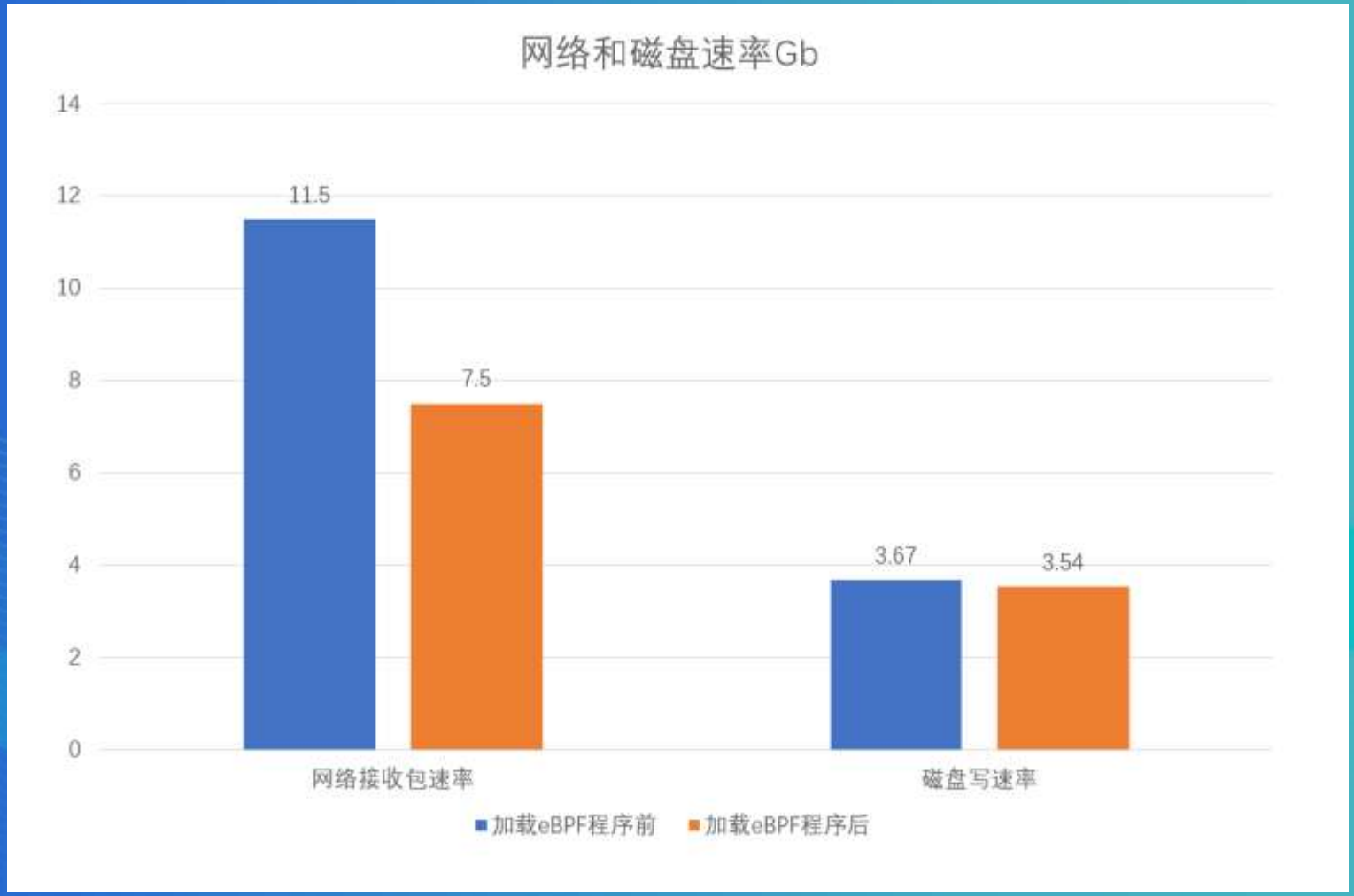
中国·西安

1. eBPF性能开销测试

- 方案：
加载BCC官方中的六个小工具（涵盖CPU、内存、IO、网络），带来多少性能消耗?

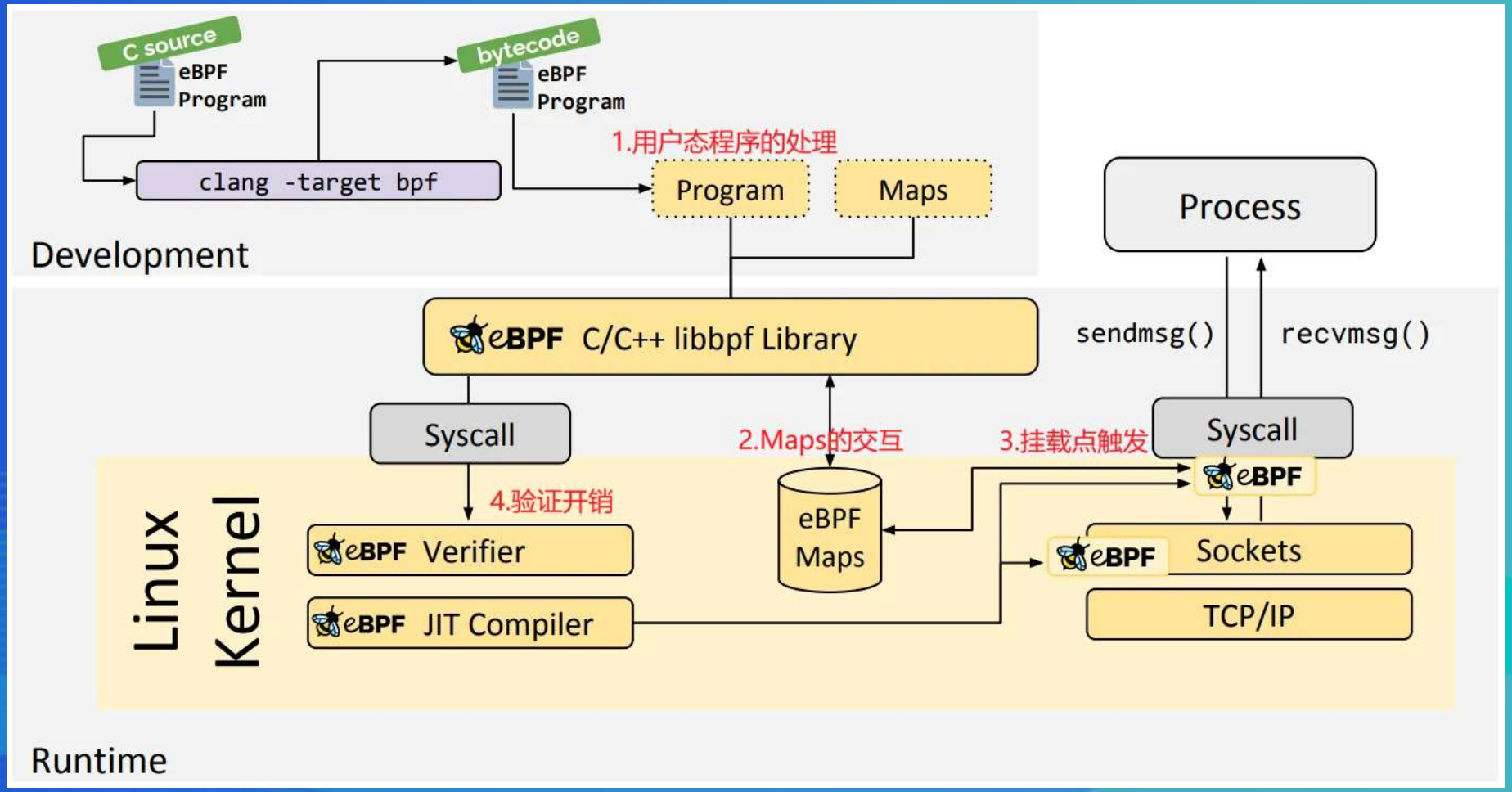
- 测试机配置：
虚拟机：ubuntu 22.04.3 LTS
CPU：13th Gen Intel(R) Core(TM) i9-13900HX 2.20 GHz 16核
内存：16GB
内核版本：6.2

- 加压工具：
stress-ng对系统加压，模拟高负载环境



2. eBPF性能开销来源分析

- 用户态处理开销
- Maps的存取开销
- Hook触发开销
- 验证器验证开销
- 内核态程序执行开销
- ……



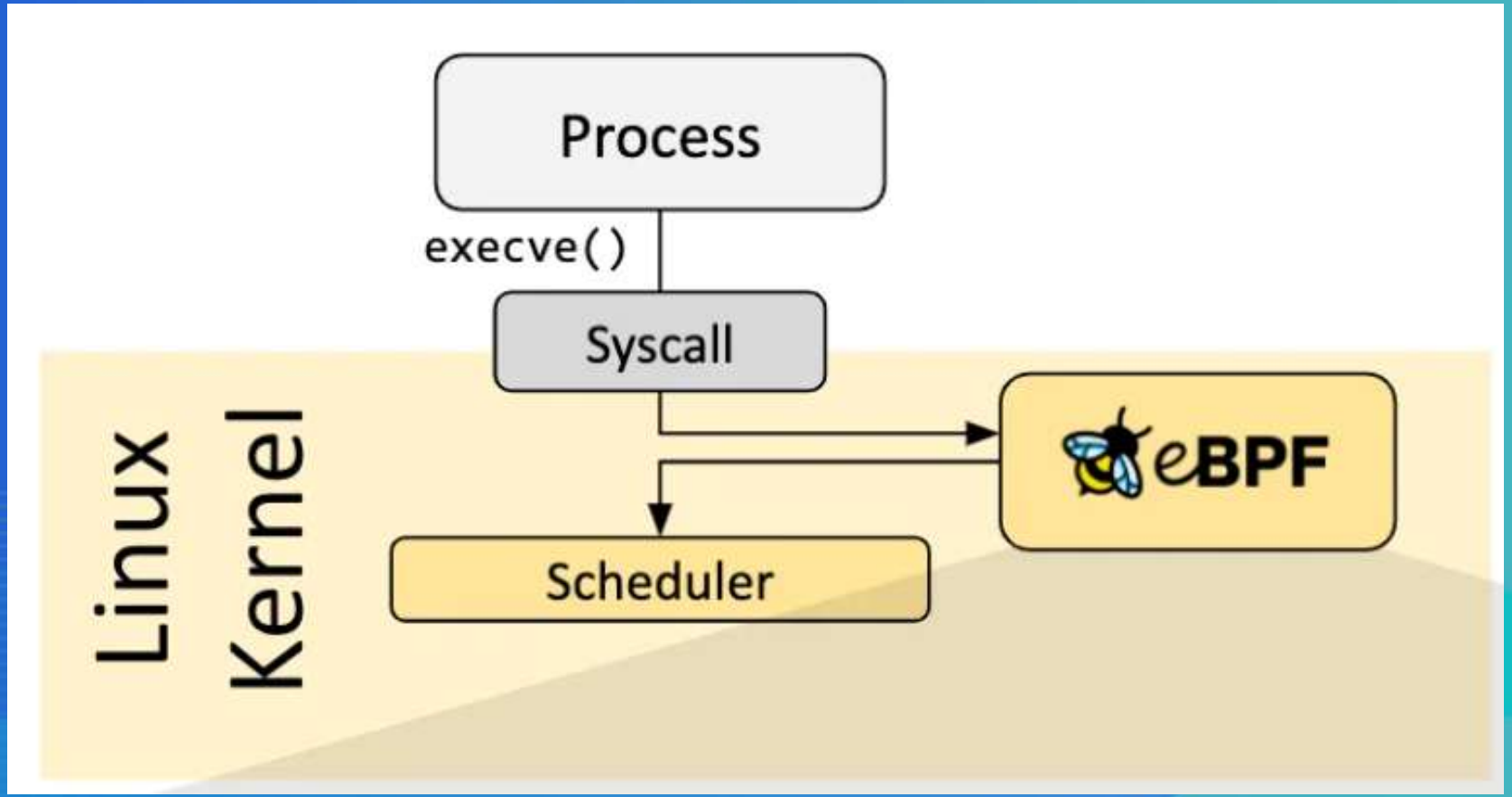
② HOOK 类型解析

1. eBPF中Hook类型

常用的Hook方式：

- kprobe/kretprobe
- tracepoint
- fentry/fexit
- XDP
- perf events
- Socket filters

。。。。。。



2. Kprobe


```c
int register_kprobe(struct kprobe *p)
{
    ret = prepare_kprobe(p);

    hlist_add_head_rcu(&p->hlist,
                &kprobe_table[hash_ptr(p->addr, KPROBE_HASH_BITS)]);

    ret = arm_kprobe(p);
}
```

```c
void arch_arm_kprobe(struct kprobe *p)
{
        u8 int3 = INT3_INSN_OPCODE;

        text_poke(p->addr, &int3, 1);
        text_poke_sync();
        perf_event_text_poke(p->addr, &p->opcode, 1, &int3, 1);
}
```

解析：

kprobe/kretprobe通过在指定地址插入 **int3软件断点异常**，CPU 触发调试异常，查表找到对应的处理程序，然后执行eBPF程序。

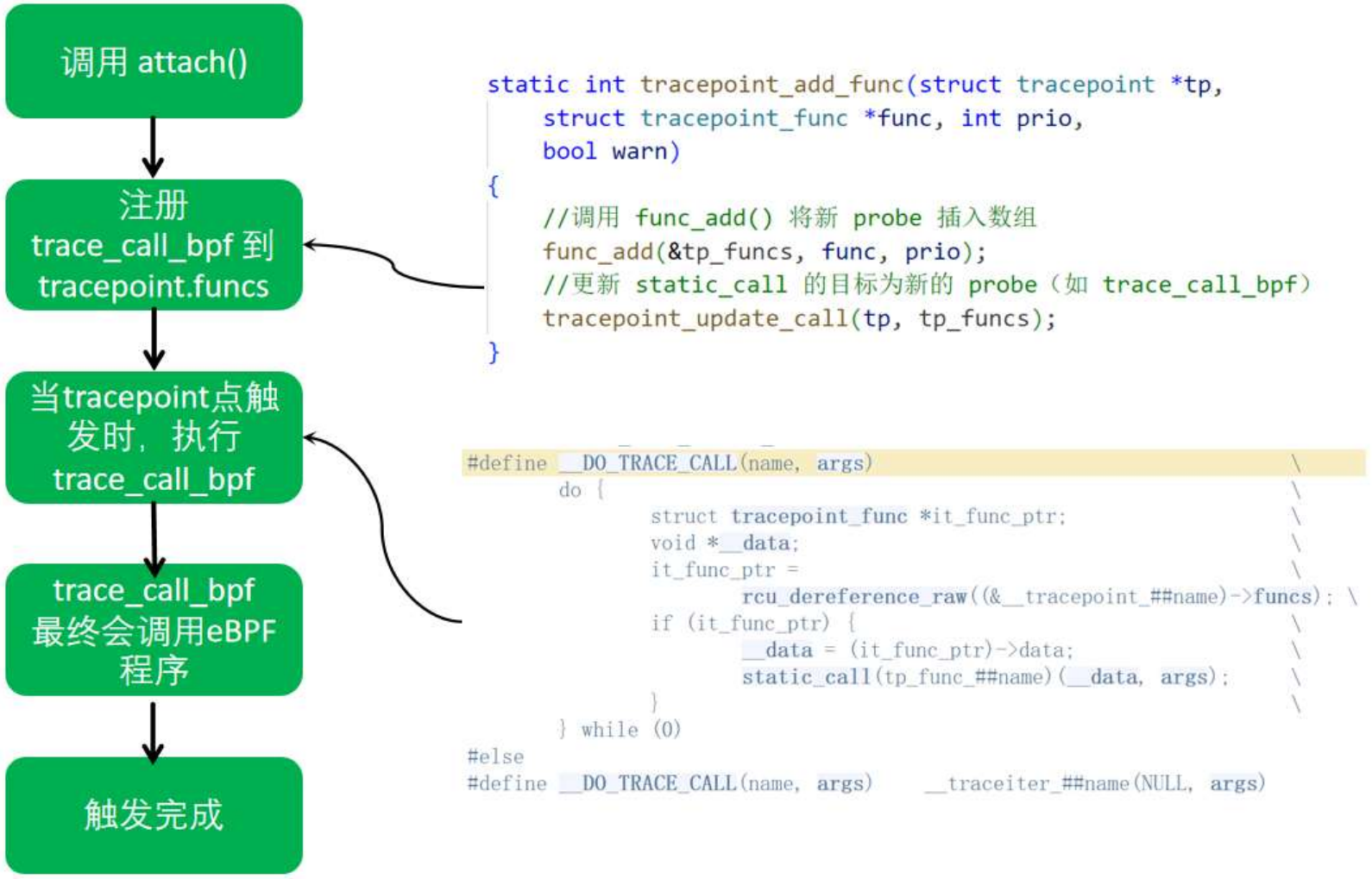
优点：

1.动态插桩
2.可插装位置多

缺点：

1.触发 int3 异常中断、查表 、栈切换。**CPU开销高**
2.内核版本依赖

3. tracepoint




```
static int tracepoint_add_func(struct tracepoint *tp,
    struct tracepoint_func *func, int prio,
    bool warn)
{
    //调用 func_add() 将新 probe 插入数组
    func_add(&tp_funcs, func, prio);
    //更新 static_call 的目标为新的 probe（如 trace_call_bpf）
    tracepoint_update_call(tp, tp_funcs);
}
```

```
#define __DO_TRACE_CALL(name, args)                                    \
        do {                                                           \
                struct tracepoint_func *it_func_ptr;                   \
                void *__data;                                          \
                it_func_ptr =                                          \
                        rcu_dereference_raw((&__tracepoint_##name)->funcs); \
                if (it_func_ptr) {                                     \
                        __data = (it_func_ptr)->data;                  \
                        static_call(tp_func_##name)(__data, args);     \
                }                                                      \
        } while (0)
#else
#define __DO_TRACE_CALL(name, args)     __traceiter_##name(NULL, args)
```

解析：

tracepoint 在内核函数中静态嵌入，通过静态函数指针调用回调函数，触发时直接执行 trace_call_bpf，无需中断、无需查表，开销极小

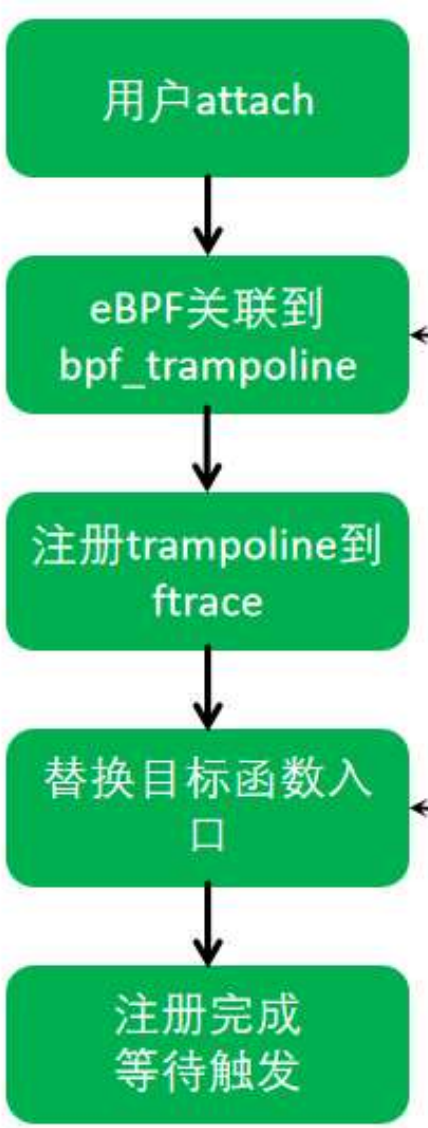
优点：

无中断、无栈切换、**原地调用**。
适配性强

缺点：

不灵活

4. fentry




```c
static int __bpf_trampoline_link_prog(struct bpf_tramp_link *link,
        struct bpf_trampoline *tr)
{
    //获取attach类型
    kind = bpf_attach_type_to_tramp(link->link.prog);
    //将ebpf程序添加到trampoline的链接列表上
    hlist_add_head(&link->tramp_hlist, &tr->progs_hlist[kind]);
    //替换call指令,确保新的eBPF生效
    err = bpf_trampoline_update(tr, true /* lock_direct_mutex */);
}
```

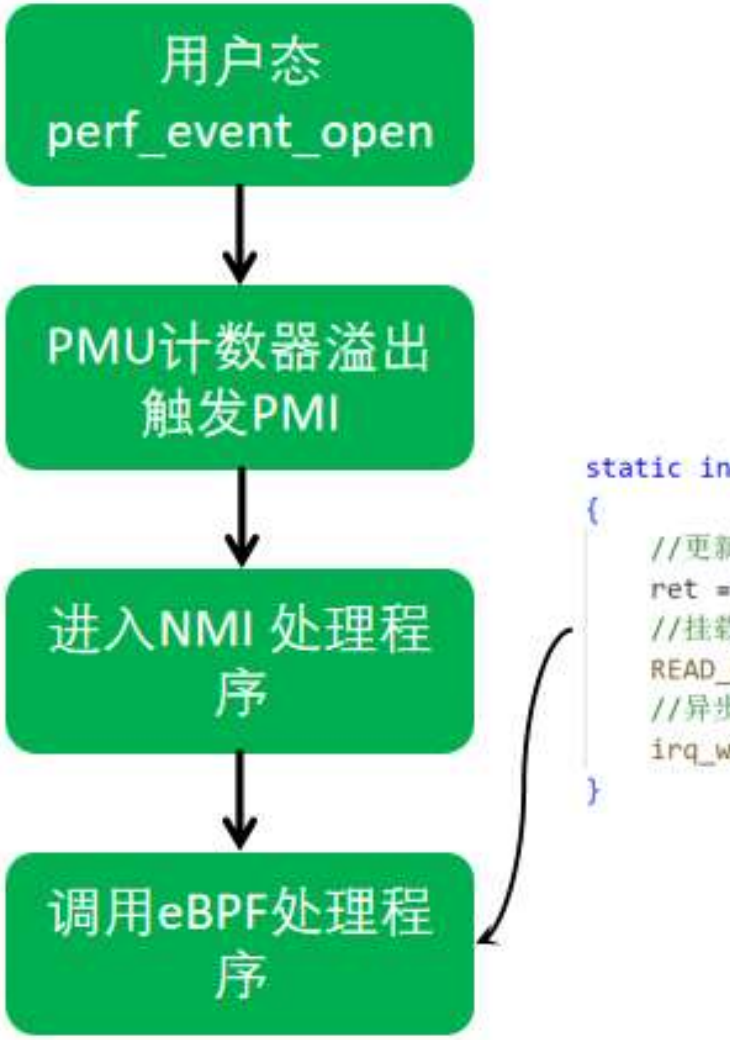
```c
static int __bpf_arch_text_poke(void *ip, enum bpf_text_poke_type t,
    void *old_addr, void *new_addr)
{
    u8 old_insn[X86_PATCH_SIZE];
    u8 new_insn[X86_PATCH_SIZE];
    prog = new_insn;
    //生成新的指令
    emit_call(&prog, new_addr, ip) :
    emit_jump(&prog, new_addr, ip);
    //替换为新指令
    text_poke_bp(ip, new_insn, X86_PATCH_SIZE, NULL);
}
```

解析：

在启用 CONFIG_FTRACE 的 x86 架构下，GCC 会在每个函数入口处插入一个 5 字节的 **NOP** 指令（call __fentry__ 的占位），内核在运行时可以用这一位置来 patch call trampoline，从而让 ftrace 或 fentry 生效。

函数入口直接调用 BPF trampoline，无中断、无异常、无查表，几乎无感知开销.
依赖高版本内核

5. perf_event




```
static int __perf_event_overflow()
{
    //更新统计
    ret = __perf_event_account_interrupt(event, throttle);
    //挂载的 BPF 程序入口
    READ_ONCE(event->overflow_handler)(event, data, regs);
    //异步通知
    irq_work_queue(&event->pending_irq);
}
```

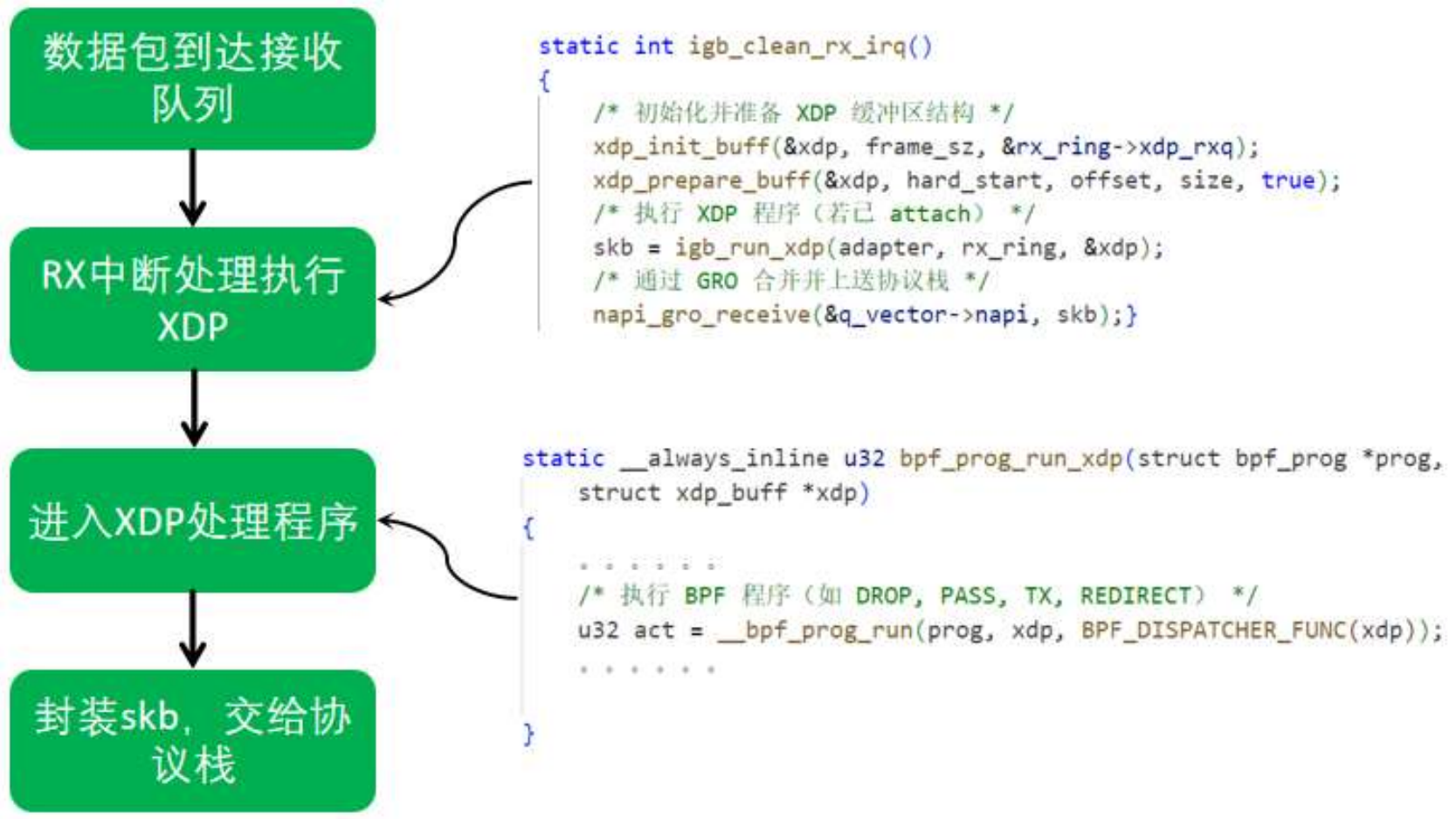
解析：

由于perf框架也支持了eBPF，
eBPF程序的能力得以增强：
1.周期性采样
2.硬件计数器触发
3.perf 数据通道

traceing类eBPF程序无法使用bpftimer进行周期性采样，因此，结合perf_event可以解决这个问题。

```
libbpf: prog 'handle_sys_enter': -- BEGIN PROG LOAD LOG --
tracing progs cannot use bpf_timer yet
```

6. XDP



解析：

XDP 是在驱动层、DMA 完成后、协议栈之前执行的超低延迟数据路径，它能在 NAPI poll 阶段即时对接收到的数据包做出决策，极大提升了网络性能与可控性。

优点：

1.驱动层处理数据包，低延迟

2.**拦截点非常早**，可快速 DROP、TX、REDIR

缺点：

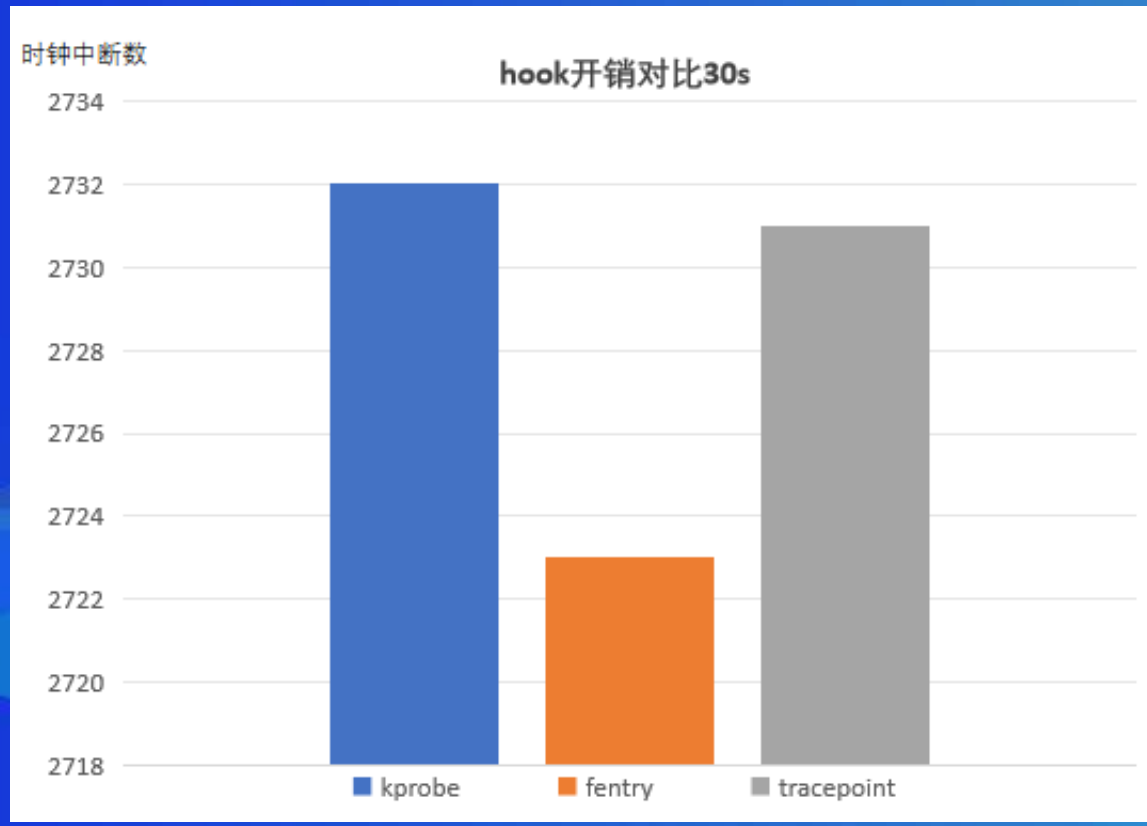
如果程序逻辑太复杂，会抵消掉“早期处理”的性能优势

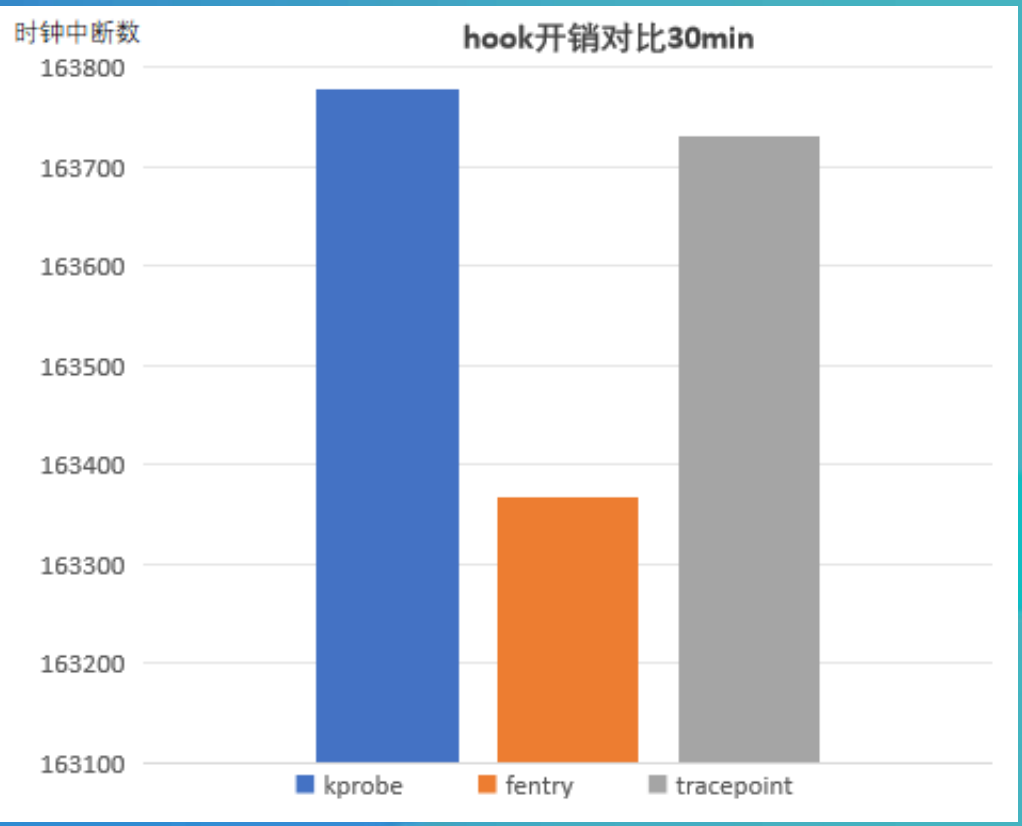
受到 verifier 限制

7. 性能开销对比测

- 统一挂载点到open系统调用入口处
- 统一三个程序只采集PID和进程名
- 其他变量都一样，只改变挂载方式

测试程序：循环执行open系统调用，触发open
衡量：记录测试程序的内核态运行时间（jiffies）

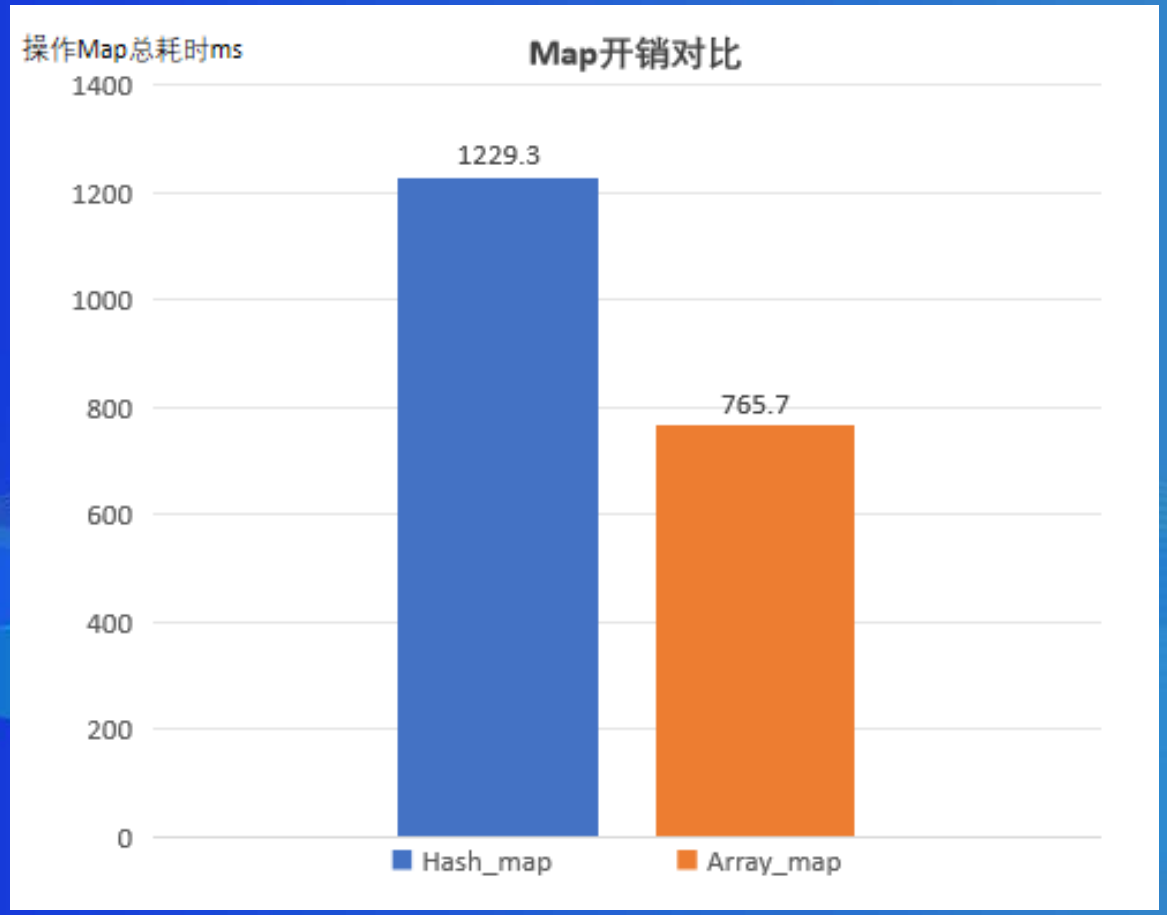




③ Map 类型源码解析与实测

1.Map类型选取

场景：在固定 key范围为（0-1024） + 高频更新的场景下。保持其他环境不变，只改变map类型为array和hash，触发1千万次更新，查看时间消耗：



这种场景下，选取Array类型的map性能更好。
但如果key值不是固定key的范围，而是pid呢?
array类型将会申请大量的内存空间，并且也不切实际！
因此，选取合适的Map也是十分重要的！

2. arraymap和per-cpu-arraymap

- array_map的查找操作是直接根据 index 计算偏移地址，访问非常快：

```
static void *array_map_lookup_elem(struct bpf_map *map, void *key)
{
    u32 index = *(u32 *)key;
    struct bpf_array *array = container_of(map, struct bpf_array, map);
    return array->value + (index * array->elem_size);
}
```

- 如果更新操作是per-cpu，会通过pptrs指针数组，读取到index对应的per-cpu数据集合，接下来通过this_cpu_ptr来获取当前CPU的value。
- array_map的更新操作是直接根据 index 计算偏移地址。

```
static int array_map_update_elem(*map, *key, *value,
    u64 map_flags)
{
    //如果是per-cpu型
    if (array->map.map_type == BPF_MAP_TYPE_PERCPU_ARRAY) {
        val = this_cpu_ptr(array->pptrs[index & array->index_mask]);
        copy_map_value(map, val, value);
    }
    else {//普通型
        val = array->value +
        (u64)array->elem_size * (index & array->index_mask);
        copy_map_value(map, val, value);
    }
}
```

3. hashmap和per-cpu-hashmap

- hash_map和per-cpu-hash的更新操作如下：

```
static int htab_map_update_elem(*map, void *key, void *value)
{
    //计算hash值
    hash = htab_map_hash(key, key_size, htab->hashrnd);

    b = __select_bucket(htab, hash);
    //普通替换
    copy_map_value_locked(map,
                  l_old->key + round_up(key_size, 8),
                  value, false);
    //普通新增
    hlist_nulls_add_head_rcu(&l_new->hash_node, head);

    //per-cpu的替换
    pcpu_copy_value(htab, htab_elem_get_ptr(l_old, key_size),
                  value, onallcpus);
}
```

- 1.hash值的计算是通过jhash( )函数计算的。
- 2.通过hash值选桶。
- 3.若是普通hash，则进行key值检索并替换。
- 4.若是per-cpu，使用pcpu_copy_value。

```
static inline u32 htab_map_hash
(const void *key, u32 key_len, u32 hashrnd)
{
    return jhash(key, key_len, hashrnd);
}
```

4.Map的内存会预先分配吗?

- array_map由于是数组结构，在程序加载的时候会直接分配需要的内存空间。
- hash_map默认也是会预先分配好内存空间，但是用户态可以配置。

```
if (prealloc) {
    if (old_elem) {
        /* if we're updating the existing element,
         * use per-cpu extra elems to avoid freelist_pop/push
         */
        pl_new = this_cpu_ptr(htab->extra_elems);
        l_new = *pl_new;
        htab_put_fd_value(htab, old_elem);
        *pl_new = old_elem;
    } else {
        struct pcpu_freelist_node *l;
        l = __pcpu_freelist_pop(&htab->freelist);
        if (!l)
            return ERR_PTR(-E2BIG);
        l_new = container_of(l, struct htab_elem, fnode);
    }
```

hash_map开启预分配时：
1.更新时，会从CPU的**备用槽**拿出一块htab_elem内存。接着把旧的元素塞回当前CPU 的备用槽。（避免频繁操作freelist）
2.插入时，从预先分配的**freelist**拿取一个空槽

当 Map 需要申请的内存空间很大、系统**内存资源很紧张**时，怎么办?

5. 关闭预分配，减少内存消耗

- hash_map和per-cpu-hash的更新操作如下：hash_map默认预先分配好内存空间，用户态可以通过设置map_flags为BPF_F_NO_PREALLOC，关闭预分配

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 10240);
    __type(key, u32);
    __type(value, u64);
    __uint(map_flags, BPF_F_NO_PREALLOC);
} hash_map SEC(".maps");
```

关闭预分配前：

```
626: hash  name hash_map  flags 0x0
        key 4B  value 8B  max_entries 10240  memlock 931712B
        btf_id 1846
```

关闭预分配后：

```
629: hash  name hash_map  flags 0x1
        key 4B  value 8B  max_entries 10240  memlock 340864B
        btf_id 1855
```

④ 内容总结

1. 降低eBPF性能开销的措施

1.选取合适的Hook点：提升采样质量、降低系统干扰

2.选择合适的 Hook 类型：fentry、tracepoint 更轻量，避免使用代价大的 hook

3.选择合适的 Map 类型与大小设置：避免 memlock 冗余

4.编写合理的过滤功能：尽早过滤不相关事件，避免无效数据处理和 map 插入

5.合理拆分 BPF 程序 / 使用 Tail Call：模块化设计、突破指令限制、提升可维护性

6.调整合适的采集频率：平衡数据精度与 overhead

eBPF 程序的性能优化是多维度的：从 hook 选择、map 类型、采样频率到输出控制、辅助特性、BPF 栈优化，每一处小细节都可能为生产环境带来数量级的资源收益。

2. Map应用场景

Map类型	应用场景	优势	劣势
BPF_MAP_TYPE_HASH	• 适合需要频繁插入的数据结构。 • 例如：存储进程信息、网络连接状态等。	• 具备良好的查找和插入性能。 • 数据量不大时，哈希表的查找时间几乎是常数级别的。	• 当哈希冲突较多时，性能会受到影响。 • 在高并发场景下，冲突更大。
BPF_MAP_TYPE_ARRAY	• 适用于需要快速随机访问的数据存储。 • 例如：访问某些状态信息，时间信息等。	• 查找性能非常优异。 • 数组查找只需要通过索引直接访问。	• 缺乏灵活性。数组的大小是固定的。 • 适合固定数据量的场景。
BPF_MAP_TYPE_PERCPU_HASH	• 为每个CPU提供单独的哈希表存储，减少多核系统上的锁竞争。 • 适用于每个CPU都有独立统计数据或状态的场景。	• 避免了并发访问的锁争用问题，提高了并发性能。	• 占用更多内存，因为为每个CPU分配了独立的哈希表。 • 如果系统有大量CPU，内存开销会显著增加。 • 数据进行统计或合并时会增加开销。
BPF_MAP_TYPE_PERCPU_ARRAY	• 每个CPU有独立的数组存储，适合频繁读取和写入的场景。 • 可以避免CPU之间的锁竞争。	• 类似于标准数组，查找和插入操作非常快，提高了并发性能。	• 占用更多内存，特别是在有多个CPU的情况下。 • 数据合并时、读取数据时需要遍历每个CPU的数组。

3. Hook方式总结

Hook类型	应用场景	优势	劣势
Tracepoint	• 稳定的系统级监控 • 跟踪cgroup相关事件 • 跟踪系统调用	• 接口稳定 • 性能开销低 • 参数结构明确	• 仅能使用内核预定义的跟踪点 • 事件参数可能不包含需要的信息
Fentry	• 高性能网络处理 • 低延迟指标收集	• 性能接近原生代码 • 直接访问函数参数 • 支持入口/退出点追踪	• 需要内核5.5+ • 依赖内核调试符号 • 架构相关
Perf_event	• CPU性能分析 • 硬件事件采集（如缓存未命中） • 调用链追踪	• 支持硬件性能计数器 • 灵活配置采样频率	• 高采样率导致高CPU开销 • 用户态数据处理复杂
XDP	• 快速丢弃攻击包 • 直接转发流量	• 处理延迟极低 • 直接访问原始网络包	• 仅限网络数据处理 • 需要驱动支持 • 无法访问完整协议栈信息
Kprobe	• 动态内核追踪 • 跟踪未导出符号的内核函数	• 可跟踪任意内核函数 • 支持函数入口(kprobe)和返回点(kretprobe)	• 性能开销大 • 不同内核版本函数签名可能变化

感谢倾听!

欢迎一起深入探讨 eBPF 性能开销!

邮箱：yys2020haha@163.com