项目信息

1.项目名称：

eBPF性能评估：多维度分析与最佳实践指导

2.方案描述：

本次eBPF性能测试的主要目标是评估不同类型的eBPF Map在各种系统资源消耗下的表现，并分析eBPF对整体系统性能的影响。通过本次测试，旨在深入了解eBPF在高负载场景下的资源开销，以及如何根据不同应用场景选择合适的 eBPF Map类型。

2.1 测试目标

- **性能指标分析**：通过测试不同Map类型（如`Hash Map`、`Array Map`、`Per-CPU Hash`等）的性能表现，评估其在高负载下对系统资源（CPU、内存、I/O等）的影响。
- **系统资源消耗评估**：监控eBPF程序运行时对系统资源的消耗情况，分析eBPF对CPU、内存、网络流量等方面的影响。
- **性能优化建议**：根据测试结果，给出合理的优化建议，指导在不同场景下使用最适合的eBPF Map类型和实现方案。

2.2 测试环境

- **硬件环境**：虚拟机配置为8核CPU、8GB内存、50GB硬盘。
- **软件环境**：操作系统为LonganOS（龙蜥操作系统），内核版本为5.x。测试工具包括`stress-ng`、`perf`、以及其他系统监控工具。
- **测试对象**：主要测试了eBPF程序在使用不同Map类型时的性能表现，包括插入、查找、删除等操作。

2.3 测试内容

- **Map类型对比**：重点对比了`Hash Map`、`Array Map`、`Per-CPU Hash Map`等不同类型的Map在高并发下的性能表现。
- **系统资源占用**：监控CPU利用率、内存使用率、磁盘I/O以及网络流量等系统指标，分析eBPF Map的操作对系统资源的影响。
- **延迟和吞吐量**：评估每种Map类型在处理不同数据规模下的延迟和吞吐量，测试其在高负载下的稳定性和性能瓶颈。

3.时间规划：

时间	任务	产出
7.20-8.2	详细设计出测试方案，对后面做出详细的规划	输出详细的测试方案
Map：		

时间	任务	产出
8.3-8.9	对Map的各个类型进行详细的理论分析	输出详细的Map类型分析报告并给出分析结论
8.10-8.16	编写测试代码并从时间维度，不同负载的情境下对不同Map进行测试	输出详细的测试结果
8.17-8.23	完善测试并对结果进行分析	输出测试结果和理论分析，并不同版本的Map特性，结合理论分析给出一个操作指南
eBPF程序对系统资源的影响：		
8.24-8.30	对eBPF程序的资源消耗进行详细的理论分析	输出详细的分析报告并给出分析结论
8.31-9.6	编写测试代码并收集在加载eBPF前后，系统整体的一个消耗情况。不同负载的情境下对不同挂载点进行测试	输出详细的测试结果
9.7-9.13	完善测试并对结果进行分析	输出测试结果和理论分析，并给出不同eBPF的特性，结合理论分析给出一个操作指南
测试补充：		
9.14-9.20	通过测试过程中发现的问题和遗漏，再补充一些需要的测试结果	初步输出一个项目总体测试文档，并查漏补缺
9.21-9.30	完善项目总体开发测试文档	输出最终的测试报告并整理项目代码

项目进度

1.已完成的工作和成果：

1.1测试不同类型的Map：

1.首先，我们需要在测试之前确定一些环境因素：

- 确定内核版本。
- 确定负载，本次测试将负载定为两大类，分别为：对系统进行加压负载、对Map的操作次数进行设置。
- 确定测试指标，本次测试会对不同Map类型定义相同的空间大小，并且进行相同的操作，来测试这些不同的Map类型在各种环境都确定的情况下，它们在时间维度上的差异。

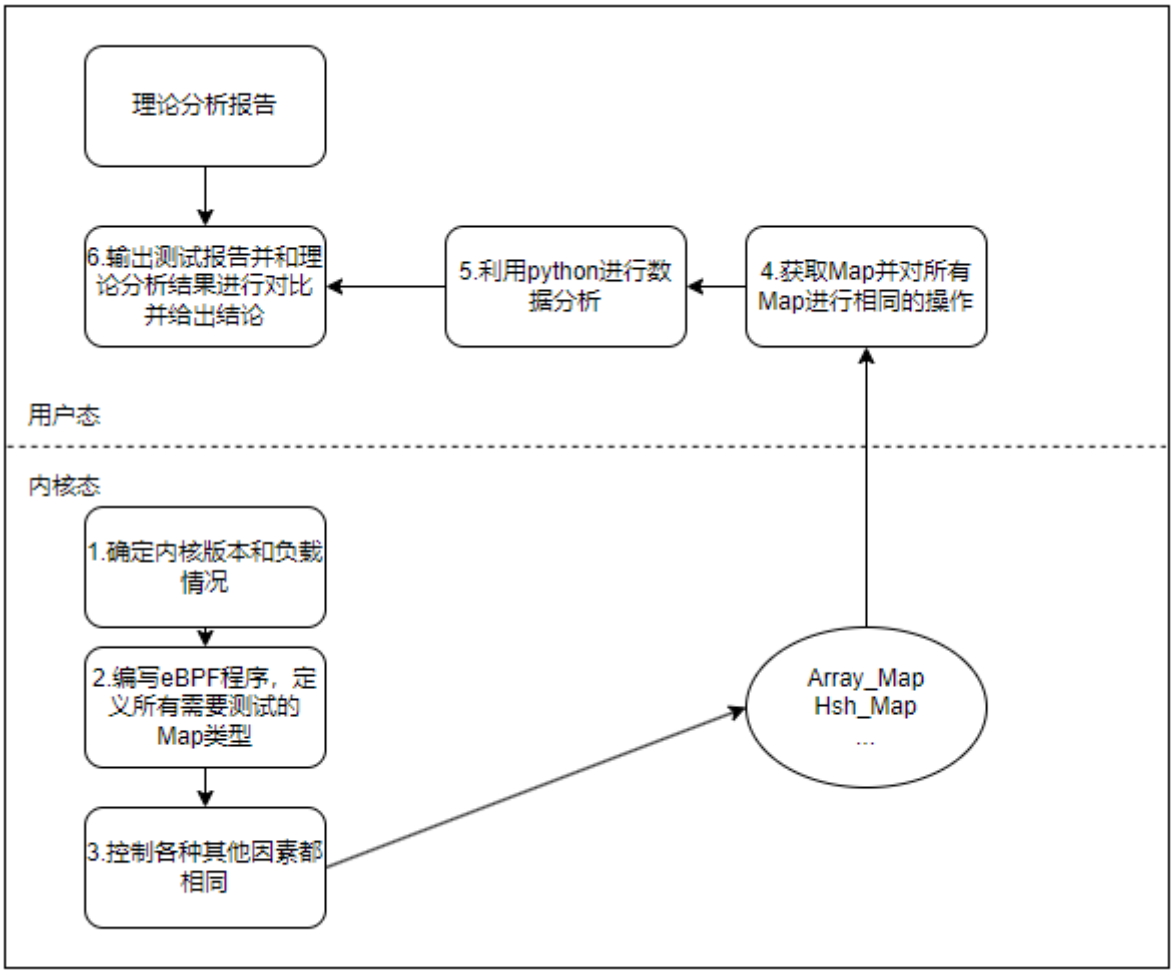
2.接下来就是编写测试程序，这里会使用libbpf来编写，编写程序的关键点为：

- 根据前面分析的结果，定义选定内核版本需要测试的Map类型。
- 将定义好的所有Map结构体挂载在同一个函数上。
- 在用户态对这些定义的Map进行获取并进行相同次数的相同操作。
- 将每个Map类型的操作时间记录下来。
- 循环上述操作，获取多组测试数据。

3.最后，编写python脚本来对得出的操作时间进行数据分析，并编写shell脚本进行上述所有操作的整合，自动化测试程序：

- 通过测试程序输出的时间数据，进行数据分析。
- 将分析的结果以文件和图表的方式展现出来。
- 输出测试报告并结合理论分析的结果给出一个在Map方面的最佳实践指南。

通过上述的描述，接下来给出一个流程图来说明本次测试的具体过程：



测试用例如下：

用例1：

事项	内容

事项	内容
场景	在系统CPU高负载情况下（CPU使用率约为80%）并且CPU频率固定，针对不同类型的eBPF Map进行相同次数的增删改查操作，这里会多次变化操作次数来说明实验结果的正确性。查看每种类型的Map在CPU高负载的情况下对于不同操作次数的耗时情况。
测试目的	评估不同类型的eBPF Map在CPU高负载的情况下，进行相同次数的CRUD操作时的性能表现，特别是记录每种类型的Map在处理时间上的差异。
负载压力产生方法	使用stress-ng来对CPU进行加压，并且每次测试时，通过设置不同的操作次数来对ebpf程序进行CRUD的压力控制。
执行脚本	map_difference_01.py,测试代码和脚本见下文所示。
执行方法	执行./run_ebpf_and_process.sh脚本；查看分析结果
与生产环境差异	测试环境为隔离的虚拟机，实际的CPU核心数要比生产环境少，并且在负载压力产生方面，也和生产环境有差异。
指标要求	每次测试之前，要控制好每种Map类型的CRUD操作次数必须相同；相同操作次数下，不同Map类型的耗时差异不应超过理论分析的预期范围；
测试结果	记录每种Map类型的CRUD操作时间，并汇总到报告中，生成柱状图或折线图展示不同Map类型的性能差异。
测试结果分析	通过测试结果的内容，并结合python的数据分析能力，来分析出在高CPU负载的情况下，哪种Map类型更适合使用，并且验证理论分析的结果是否正确。
后续Action	将详细的测试分析结果编写到文档中，并且给出一个不同Map类型的适用场景的有力指导。

用例2：

事项	内容
场景	在系统内存高负载情况下（内存使用率约为75%）并且CPU频率固定，针对不同类型的eBPF Map进行相同次数的增删改查操作，这里会多次变化操作次数来说明实验结果的正确性。查看每种类型的Map在内存高负载的情况下对于不同操作次数的耗时情况。
测试目的	评估不同类型的eBPF Map在内存高负载的情况下，进行相同次数的CRUD操作时的性能表现，特别是记录每种类型的Map在处理时间上的差异。
负载压力产生方法	使用stress-ng来对内存进行加压，并且每次测试时，通过设置不同的操作次数来对ebpf程序进行CRUD的压力控制。

事项	内容
执行脚本	map_difference_02.py。测试代码和脚本见下文所示。
执行方法	执行./run_ebpf_and_process.sh脚本；查看分析结果
与生产环境差异	测试环境为隔离的虚拟机，实际的内存总大小要比生产环境小，并且在负载压力产生方面，也和生产环境有差异。
指标要求	每次测试之前，要控制好每种Map类型的CRUD操作次数必须相同；相同操作次数下，不同Map类型的耗时差异不应超过理论分析的预期范围；
测试结果	记录每种Map类型的CRUD操作时间，并汇总到报告中，生成柱状图或折线图展示不同Map类型的性能差异。
测试结果分析	通过测试结果的内容，并结合python的数据分析能力，来分析出在内存高负载的情况下，哪种Map类型更适合使用，并且验证理论分析的结果是否正确。
后续Action	将详细的测试分析结果编写到文档中，并且给出一个不同Map类型的适用场景的有力指导。

用例3：

事项	内容
场景	在系统开启CPU的P-states和C-states时，针对不同类型的eBPF Map进行相同次数的增删改查操作，这里会多次变化操作次数来说明实验结果的正确性。查看每种类型的Map在系统开启CPU的P-states和C-states的情况下对于不同操作次数的耗时情况。
测试目的	评估不同类型的eBPF Map在内存高负载的情况下，进行相同次数的CRUD操作时的性能表现，特别是记录每种类型的Map在处理时间上的差异。
负载压力产生方法	使用stress-ng来对系统进行加压，并且每次测试时，通过设置不同的操作次数来对ebpf程序进行CRUD的压力控制。
执行脚本	map_difference_03.py。测试代码和脚本见下文所示。
执行方法	执行./run_ebpf_and_process.sh脚本；查看分析结果
与生产环境差异	测试环境为隔离的虚拟机，实际的CPU频率变化与生产环境有一定的差异，并且在负载压力产生方面，也和生产环境有差异。
指标要求	每次测试之前，要控制好每种Map类型的CRUD操作次数必须相同；相同操作次数下，不同Map类型的耗时差异不应超过理论分析的预期范围；

事项	内容
测试结果	记录每种Map类型的CRUD操作时间，并汇总到报告中，生成柱状图或折线图展示不同Map类型的性能差异。
测试结果分析	通过测试结果的内容，并结合python的数据分析能力，来分析出在内存高负载的情况下，哪种Map类型更适合使用，并且验证理论分析的结果是否正确。
后续Action	将详细的测试分析结果编写到文档中，并且给出一个不同Map类型的适用场景的有力指导。

用例4：

事项	内容
场景	在系统关闭CPU的P-states和C-states时，针对不同类型的eBPF Map进行相同次数的增删改查操作，这里会多次变化操作次数来说明实验结果的正确性。查看每种类型的Map在系统关闭CPU的P-states和C-states的情况下对于不同操作次数的耗时情况。
测试目的	评估不同类型的eBPF Map在内存高负载的情况下，进行相同次数的CRUD操作时的性能表现，特别是记录每种类型的Map在处理时间上的差异。
负载压力产生方法	使用stress-ng来对系统进行加压，并且每次测试时，通过设置不同的操作次数来对ebpf程序进行CRUD的压力控制。
执行脚本	map_difference_04.py。测试代码和脚本见下文所示。
执行方法	执行./run_ebpf_and_process.sh脚本，查看分析结果。
与生产环境差异	测试环境为隔离的虚拟机，实际的CPU频率变化与生产环境有一定的差异，并且在负载压力产生方面，也和生产环境有差异。
指标要求	每次测试之前，要控制好每种Map类型的CRUD操作次数必须相同；相同操作次数下，不同Map类型的耗时差异不应超过理论分析的预期范围；
测试结果	记录每种Map类型的CRUD操作时间，并汇总到报告中，生成柱状图或折线图展示不同Map类型的性能差异。
测试结果分析	通过测试结果的内容，并结合python的数据分析能力，来分析出在关闭CPU的P-states和C-states的情况下，哪种Map类型更适合使用，并且验证理论分析的结果是否正确。
后续Action	将详细的测试分析结果编写到文档中，并且给出一个不同Map类型的适用场景的有力指导。

测试的**部分代码和脚本**如下：

1.将eBPF程序挂载到系统调用入口处：

```
SEC("tracepoint/raw_syscalls/sys_enter")
int tp_sys_entry(struct trace_event_raw_sys_enter *args) {
    return analyze_maps(args,&rb,e);
}
```

2.内核态申请Map空间，并将采集的内容存入到Map中：

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 1024);//12KB
    __type(key, u32);
    __type(value,u64);
} hash_map SEC(".maps");
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 1024);
    __type(key, u32);
    __type(value,u64);
} array_map SEC(".maps");
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
    __uint(max_entries, 1024);
    __type(key, u32);
    __type(value,u64);
} percpu_array_map SEC(".maps");
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_HASH);
    __uint(max_entries, 1024);
    __type(key, u32);
    __type(value,u64);
} percpu_hash_map SEC(".maps");
//在内核态中将数据信息存入到相应的map中
volatile __u64 k = 0;
#define MAX_ENTRIES 1024
static int analyze_maps(struct trace_event_raw_sys_enter *args,void *rb,
                                struct common_event *e){
    u32 idx,counts;
    u64 syscall_id = (u64)args->id;
    // 使用原子操作递增k，并获取递增前的值
    idx = __sync_fetch_and_add(&k, 1);
    // 确保k在0到MAX_ENTRIES之间循环(避免同步问题)
    if (idx >= MAX_ENTRIES) {
        __sync_bool_compare_and_swap(&k, idx + 1, 0);
        idx = 0;
    }
    // 向hash、array类型的map中存入数据
    bpf_map_update_elem(&hash_map, &idx, &syscall_id, BPF_ANY);
    bpf_map_update_elem(&array_map, &idx, &syscall_id, BPF_ANY);
    bpf_map_update_elem(&percpu_array_map,&idx,&syscall_id,BPF_ANY);
    bpf_map_update_elem(&percpu_hash_map,&idx,&syscall_id,BPF_ANY);
    bpf_map_update_elem(&percpu_hash_map,&idx,&syscall_id,BPF_ANY);
```

```
    RESERVE_RINGBUF_ENTRY(rb, e);
    e->test_ringbuff.key = idx;
    e->test_ringbuff.value = syscall_id;
    bpf_ringbuf_submit(e, 0);
    bpf_printk("syscall_id = %llu\n", syscall_id);
    return 0;
}
#endif /* __ANALYZE_MAP_H */
```

3.用户态计算操作的时间：

```
//查找Per_cpu HashMap
    clock_gettime(CLOCK_MONOTONIC, &start);
    for (key = 1; key < MAX_ENTRIES; key++) {
        random_number = (rand() % key);
        __u64 *values = malloc(value_size);
        if (bpf_map_lookup_elem(per_cpu_hash_fd, &random_number, values) != 0) {
            fprintf(stderr, "Failed to lookup element in percpu_hash_map: %d\n",
                    errno);
            return 1;
        }
        // 根据需要处理 CPU 上的值
        for (int cpu = 0; cpu < MAX_CPUS; cpu++) {
            if (values[cpu]) {
                // 可以根据需要打印或处理每个 CPU 上的值
            }
        }
        free(values);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    elapsed = diff(start, end);
    snprintf(formatted_time, sizeof(formatted_time), "%ld.%09ld",
             elapsed.tv_sec, elapsed.tv_nsec);
    printf("%-13s", formatted_time);
    // 刷新输出缓冲区
    fflush(stdout);
```

4.使用python分析数据

```
import pandas as pd
import matplotlib.pyplot as plt
# 步骤 1: 将 .txt 文件转换为 .csv 文件
input_txt_file = './output.txt'  # 你的 .txt 文件路径
output_csv_file = './data.csv'  # 输出 .csv 文件路径
# 读取 .txt 文件内容并写入 .csv 文件
with open(input_txt_file, 'r') as txt_file:
    lines = txt_file.readlines()
# 写入 .csv 文件
with open(output_csv_file, 'w') as csv_file:
    for line in lines:
        # 替换多余空格为逗号，准备写入到 .csv 文件
```

```
        formatted_line = ','.join(line.split())
        csv_file.write(formatted_line + '\n')
# 步骤 2: 读取 .csv 文件并进行数据分析
data = pd.read_csv(output_csv_file, header=None)
# 给列命名，每三个数据为一组，分别对应 lookup、insert、delete 操作
data.columns = ['hash_lookup', 'hash_insert', 'hash_delete',
                'array_lookup', 'array_insert', 'array_delete',
                'percpu_array_lookup', 'percpu_array_insert', 'percpu_array_delete',
                'percpu_hash_lookup', 'percpu_hash_insert', 'percpu_hash_delete']
# 计算每种 map 类型的平均操作时间
avg_hash = data[['hash_lookup', 'hash_insert', 'hash_delete']].mean()
avg_array = data[['array_lookup', 'array_insert', 'array_delete']].mean()
avg_percpu_array = data[['percpu_array_lookup', 'percpu_array_insert',
'percpu_array_delete']].mean()
avg_percpu_hash = data[['percpu_hash_lookup', 'percpu_hash_insert',
'percpu_hash_delete']].mean()
# 创建一个 DataFrame 来存储平均值
avg_table = pd.DataFrame({
    'Operation': ['lookup', 'insert', 'delete'],
    'Hash Map': avg_hash.values,
    'Array Map': avg_array.values,
    'Per-CPU Array': avg_percpu_array.values,
    'Per-CPU Hash': avg_percpu_hash.values
})
# 打印平均值表格到控制台
print("Average Execution Time of eBPF Map Operations (in seconds):\n")
print(avg_table.to_string(index=False))
# 绘制平均操作时间的图表
plt.figure(figsize=(10, 6))
# 绘制四种 map 类型的平均时间曲线
operations = ['lookup', 'insert', 'delete']
plt.plot(operations, avg_hash, marker='o', linestyle='-', color='b', label='Hash')
plt.plot(operations, avg_array, marker='s', linestyle='-', color='r', label='Array')
plt.plot(operations, avg_percpu_array, marker='^', linestyle='-', color='g',
label='Per-CPU Array')
plt.plot(operations, avg_percpu_hash, marker='d', linestyle='-', color='purple',
label='Per-CPU Hash')
# 图表设置
plt.title('Average Execution Time of eBPF Map Operations')
plt.xlabel('Operation Type')
plt.ylabel('Time (seconds)')
plt.legend(loc='upper left')
plt.grid(True)
# 显示图表
plt.savefig('ebpf_map_operation_times.png')  # 保存图表为文件
plt.show()
```

测试结果：

用例1：

CPU高负载（利用率80%）：

```
stress-ng --cpu 8 --cpu-load 80 --timeout 600s
```

Operation	Hash Map	Array Map	Per-CPU Array	Per-CPU Hash
lookup	0.001882	0.001717	0.009763	0.009444
insert	0.001819	0.002745	0.007086	0.006651
delete	0.002188	0.002347	0.009136	0.006414

通过python脚本进行数据分析和处理，得到折线图：

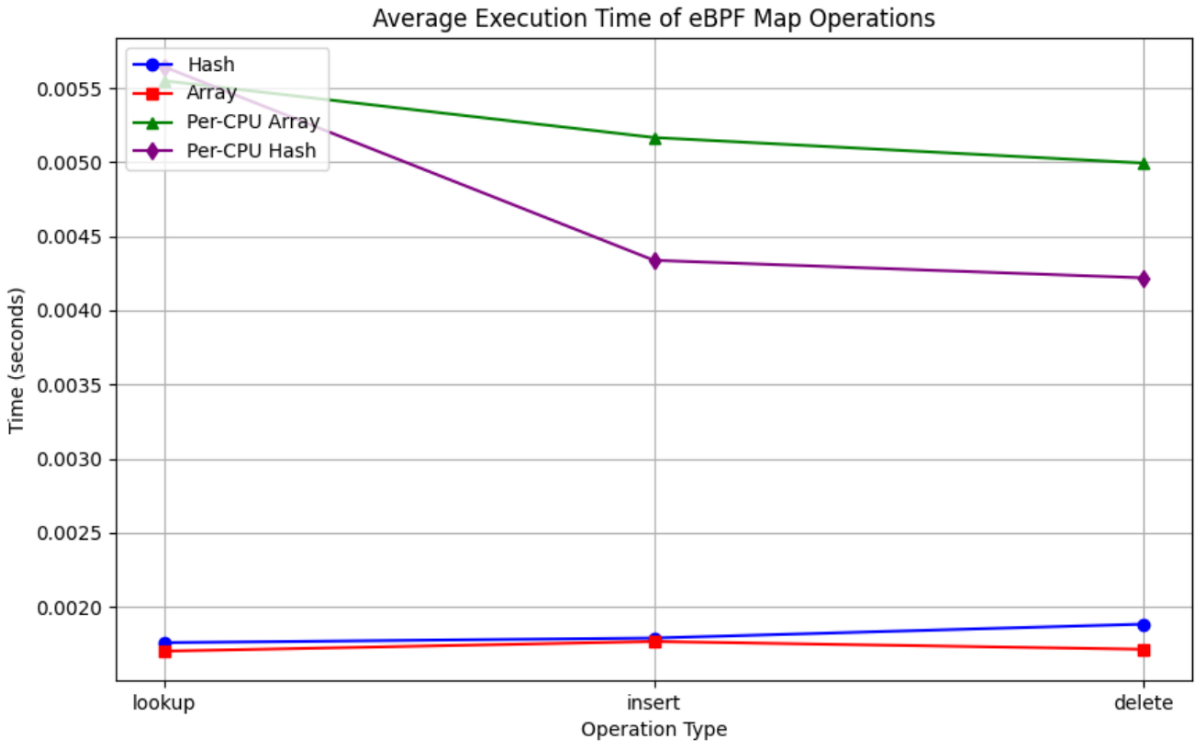
用例2：

内存高负载（利用率75%）：

```
stress-ng --vm 1 --vm-bytes 90% --timeout 600s
```

Operation	Hash Map	Array Map	Per-CPU Array	Per-CPU Hash
lookup	0.001758	0.001700	0.005549	0.005642
insert	0.001789	0.001766	0.005167	0.004338
delete	0.001883	0.001713	0.004995	0.004220

通过python脚本进行数据分析和处理，得到折线图：



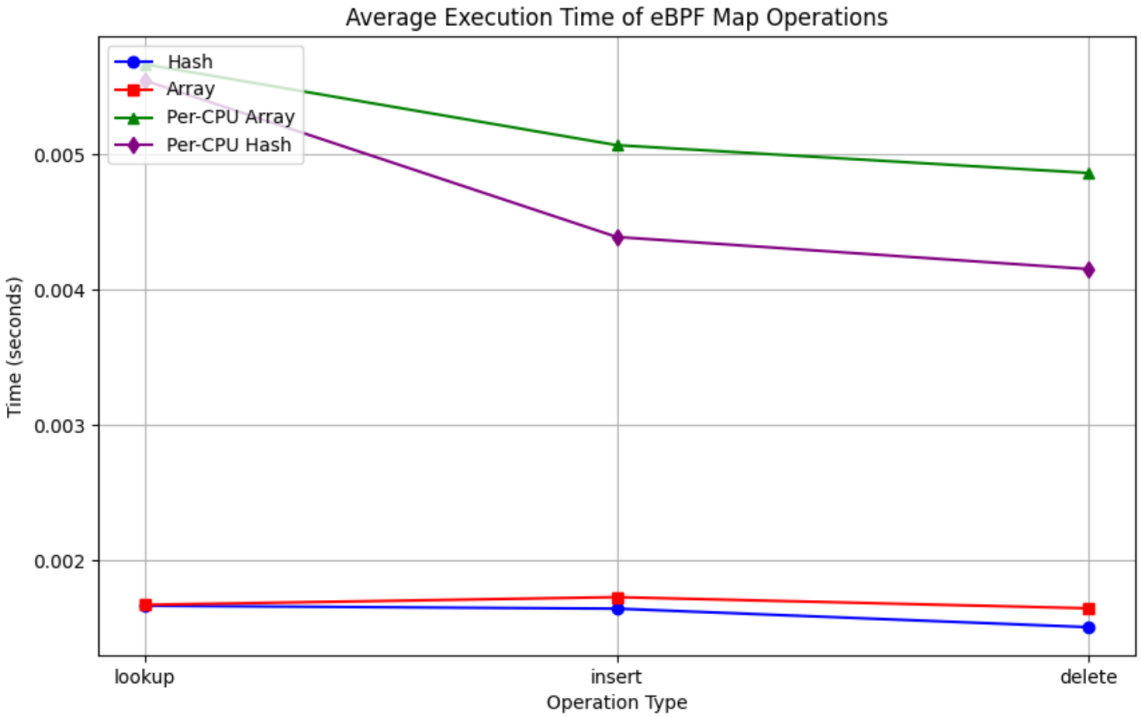
用例3和4：

由于在本人测试机上开启和关闭CPU的P-states和C-states时，差异几乎没有（因为测试机的整体性能相比生产环境小相差较大），因此我在这里只列举用例4的测试结果。

关闭CPU P-states和C-states：

```
Operation  Hash Map  Array Map  Per-CPU Array  Per-CPU Hash
   lookup  0.001664   0.001671       0.005664      0.005544
   insert  0.001643   0.001728       0.005067      0.004389
   delete  0.001506   0.001645       0.004862      0.004152
```

通过python脚本进行数据分析和处理，得到折线图：



测试结果分析（结合理论分析结果）：

BPF_MAP_TYPE_HASH：

- **应用场景**：典型的键值对存储，适合需要频繁查找和插入的数据结构。常用于存储进程信息、网络连接状态等需要快速访问的动态数据。
- 优点
  - 具备良好的查找性能，尤其是当数据量不大时，哈希表的查找时间几乎是常数级别的。
- 缺点
  - 当哈希冲突较多时，性能会受到影响。特别是在高并发场景下，哈希表可能会面临严重的冲突，导致性能下降。
  - 插入和删除操作在冲突时需要链式或者开放寻址法处理，增加了操作的复杂性和开销。
- **测试结果预期**：查找操作的平均时间应较低且稳定，但插入和删除操作的耗时可能会增加，尤其是在多线程或高并发下，随着数据量增大，性能可能逐渐退化。

BPF_MAP_TYPE_ARRAY:

- **应用场景**：适用于需要快速随机访问的数据存储。典型场景包括定长数组存储，比如为每个CPU存储统计信息等。
- 优点
  - 查找性能非常优异，因为数组查找只需要通过索引直接访问。适合频繁读取的场景。
  - 插入操作通常不需要进行复杂的哈希计算或冲突处理，因此插入速度较快。
- 缺点
  - 缺乏灵活性。数组的大小是固定的，适合小数据量的场景，难以处理动态大小的数据集。
  - 需要事先确定数组的大小，因此对内存有较大的消耗，特别是在数据量较大的情况下。

- **测试结果预期**：查找、插入和删除操作都应当表现出极低的耗时，因为它依赖于数组的直接索引访问。在多线程场景中，由于没有锁争用，性能也应保持稳定。

BPF_MAP_TYPE_PERCPU_HASH

- **应用场景**：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。
- 优点
  - 避免了在多核系统上多个CPU对同一数据结构进行并发访问的锁争用问题，提高了并发性能。
  - 每个CPU都有独立的哈希表，因此在高并发下查找和插入操作的性能表现优异。
- 缺点
  - 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。
  - 数据汇总的复杂性较高。例如，当需要对所有CPU的数据进行统计或合并时，需要手动遍历每个CPU的数据，这会带来额外的开销。
- **测试结果预期**：查找操作的性能应接近于标准的哈希表，但插入和删除操作的开销较高，尤其在多核环境下，需要考虑数据在不同CPU间的合并复杂性。

BPF_MAP_TYPE_PERCPU_ARRAY

- **应用场景**：每个CPU有独立的数组存储，适合频繁读取和写入的场景，并且可以避免CPU之间的锁竞争。
- 优点
  - 类似于标准数组，查找和插入操作非常快。
  - 每个CPU有独立的存储空间，避免了锁争用问题，特别适合高并发写入场景。
- 缺点
  - 和`BPF_MAP_TYPE_PERCPU_HASH`类似，占用大量内存，特别是在有多个CPU的情况下。
  - 数据需要合并时会带来额外开销，特别是在读取数据时需要遍历每个CPU的数组。
- **测试结果预期**：查找和插入性能非常好，几乎没有锁竞争，性能曲线应非常平稳。在数据合并时可能有一些额外开销，但由于每个CPU独立存储，这部分开销主要体现在数据读取阶段。

理论指导：

- **高并发场景**：对于需要处理大量并发写入的场景，`BPF_MAP_TYPE_PERCPU_HASH`或`BPF_MAP_TYPE_PERCPU_ARRAY`是较好的选择，能够避免锁竞争并提高并发性能。然而，需要权衡内存开销和数据汇总的复杂性。

- **只读场景**：如果系统大部分操作是只读的，`BPF_MAP_TYPE_ARRAY`或`BPF_MAP_TYPE_HASH`会更合适，它们能够提供非常快的查找操作并且内存占用较小。

-数据合并复杂性：在需要频繁合并数据的场景下，标准的Hash和Array Map可能更适合，因为Per-CPU Map虽然提高了并发性能，但数据合并的复杂性会增加程序的总体处理时间。

由于内存布局连续，数组型Map在内存访问时具有很高的CPU缓存命中率。CPU通常以缓存行（cache line）为单位进行数据加载，连续的内存布局意味着更多的相关数据可以一次性被加载到缓存中。这使得查找和更新操作的速度非常快，特别是在`BPF_MAP_TYPE_ARRAY`和`BPF_MAP_TYPE_PERCPU_ARRAY`中。

在多核系统中，Hash类Map容易出现锁争用，特别是在频繁插入和删除的场景下。即使在无冲突的情况下，多个CPU同时对同一哈希表进行读写操作也会导致锁竞争。`BPF_MAP_TYPE_PERCPU_HASH`通过为每个CPU分配独立的哈希表，解决了这一问题，但会带来内存开销和数据合并的复杂性。

1.2测试eBPF程序对系统资源的消耗：

用例设计和结果如下：

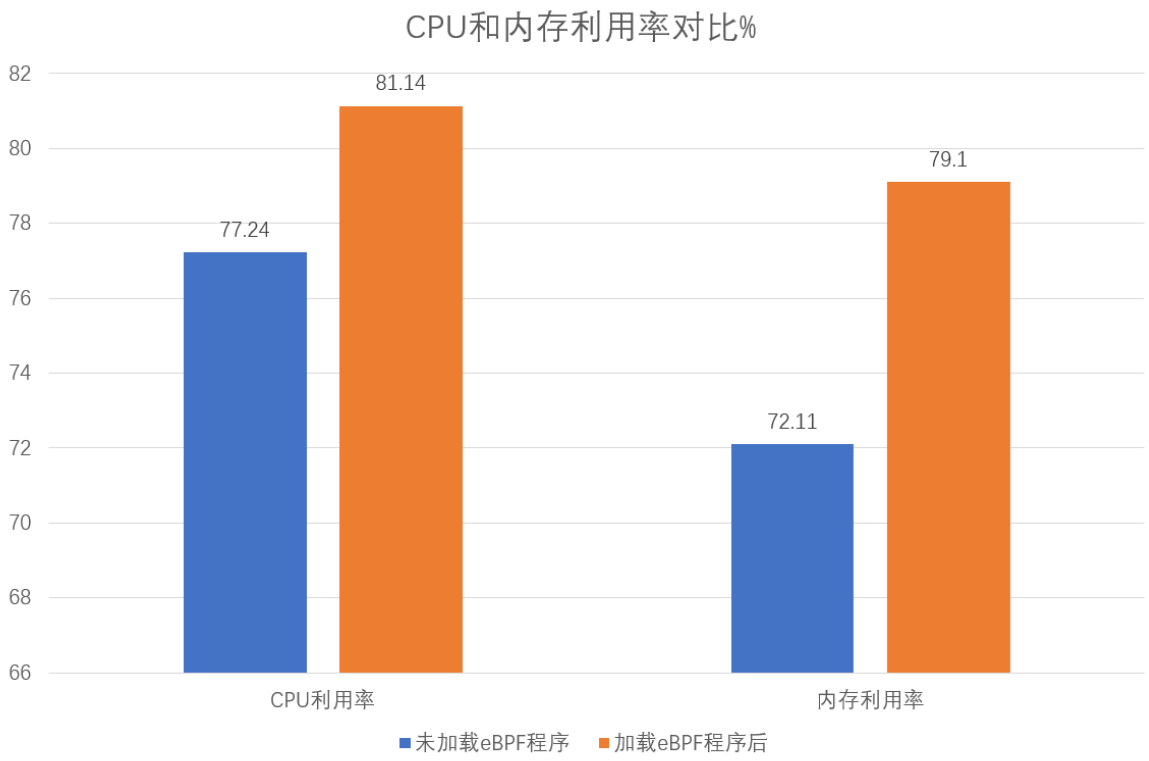
用例5：

事项	内容
场景	给系统的CPU、内存、磁盘IO、网络加压后。加载bcc工具中的biolatency、biosnoop、cpudist、offwaketime、filetop、tcpsubnet这6个eBPF工具，并且记录在加载这六个工具前后，系统的资源利用情况。
测试目的	评估加载eBPF程序对系统的资源消耗。
负载压力产生方法	使用stress-ng来对系统的CPU、内存、磁盘IO、网络进行加压。
执行脚本	resource_consume.py
执行方法	先记录未加载ebpf程序时，系统的各项资源使用情况，接下来再加载eBPF程序，记录系统的各项资源使用情况。最后执行resource_consume.py脚本，查看分析结果。
与生产环境差异	测试环境为隔离的虚拟机，实际的压力情况与生产环境有一定的差异，并且在负载压力产生方面，也和生产环境有差异。
指标要求	测试时，要基本保持系统的其他程序基本一致，尽量减少其他因素对系统的影响。
测试结果	通过Glances工具获取系统的各项指标，并且输出为.csv文件，方便后续对数据的分析。
测试结果分析	通过测试结果的内容，并结合python的数据分析能力，来分析出加载eBPF程序后，系统的资源消耗情况。
后续Action	将详细的测试分析结果编写到文档中，并且给出一个详细说明。

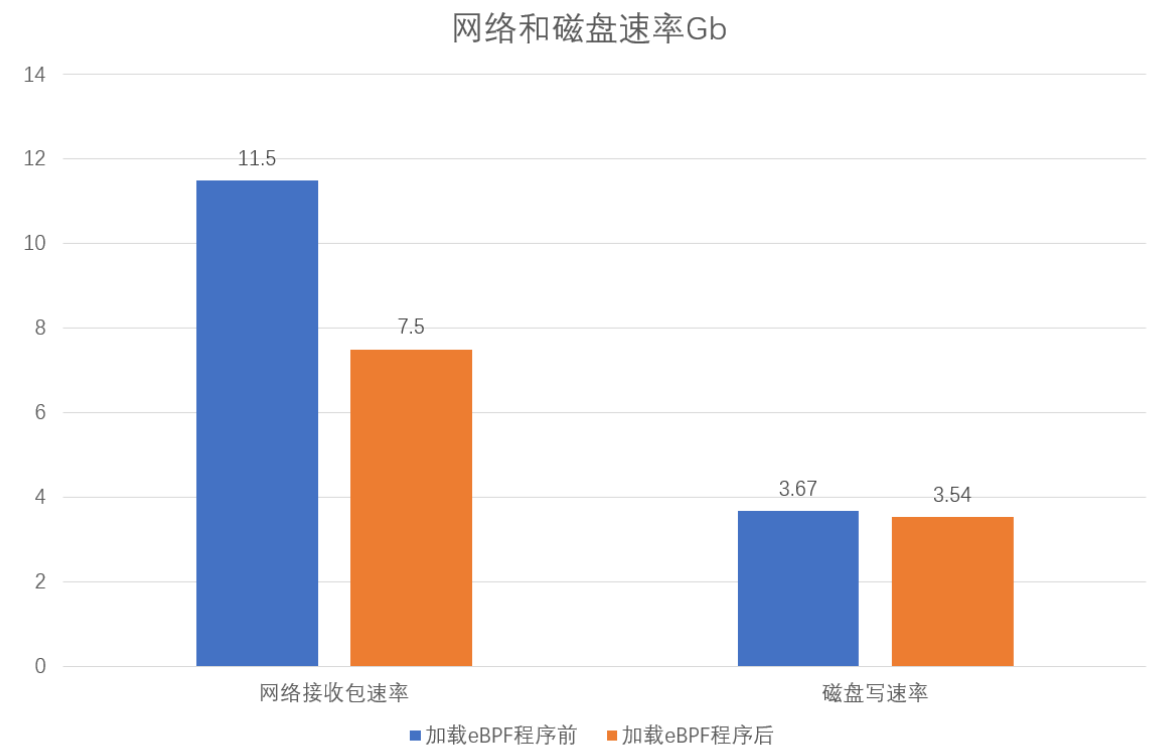
加压指令：

```
sudo stress-ng --cpu 4 --cpu-method all --timeout 1200s \
               --vm 4 --vm-bytes 7G \
               --hdd 4 --timeout 1200s \
               --sock 4 --timeout 1200s
```

1.CPU和内存利用率对比：



2.网络接收速率和磁盘写速率对比：



测试结果分析：

系统资源消耗分析：

1. **CPU 消耗**

- **上下文切换**： eBPF 程序的执行通常涉及内核与用户空间之间的上下文切换，这会增加 CPU 的负担，尤其是在高频率事件触发时（如网络数据包处理）。
- **程序复杂性**： eBPF 程序逻辑复杂，包含大量计算或条件判断，将直接消耗更多的 CPU 资源。
- **负载增加**： eBPF 程序通过附加到特定事件（如系统调用或网络包）来监控或操控系统行为，频繁的事件触发会导致 CPU 利用率上升。

2. **内存消耗**

- **内存分配**： eBPF 程序在执行过程中需要动态分配内存（如用于存储数据），这可能导致内存使用增加。
- **数据结构**： eBPF 使用的映射（如 hash 或 array）会消耗内存，特别是在数据量大时，映射的内存开销也会随之增加。
- **历史数据**： 如果程序保留历史数据（如统计信息），则需要占用更多内存，特别是在长时间运行的情况下。

3. **网络消耗**

- **数据包捕获**： eBPF 程序常用于网络监控和数据包捕获，这可能导致网络接口的 CPU 处理负担增加，尤其是在高流量情况下。
- **网络协议处理**： 对网络协议的解析和处理可能需要更多的 CPU 资源，这可能影响网络延迟和吞吐量。
- **事件频率**： 高频率的网络事件会导致大量的 eBPF 程序触发，这可能引起网络拥堵和延迟。

4. **磁盘 I/O 消耗**

- **数据记录**： 如果 eBPF 程序用于记录事件数据到磁盘（如日志记录），则会增加磁盘 I/O 操作的频率，导致更高的 I/O 消耗。
- **持久化存储**： 将收集的数据存储到持久化存储中（如数据库或文件系统）需要频繁的读写操作，增加磁盘负载。
- **映射大小**： 在使用 eBPF 映射（如 hash 映射）存储大量信息时，可能需要频繁地写入磁盘以进行持久化，尤其是在系统重启或程序重新加载时。

1.3对Map进行详细的理论分析

在进行测试之前，需要先对测试的Map类型进行一个详细的理论分析，通过对Map的理论分析，来得到一个有利的理论指导，并通过这个指导来设计测试方案并且验证测试结果的正确性。由于理论分析文档较大，这里就不展示详细内容了，在本项目的docs文档下有详细的理论分析文档。这里简要介绍一下：本项目通过分析了对eBPF中的几种常见Map类型进行了详细的理论分析，包括Hash、Array、Per-CPU Hash、Per-CPU Array和Ring Buffer。通过源码剖析，文章深入探讨了每种Map类型的实现细节、操作方式及其性能特点。特别是在高并发场景下，Per-CPU Map类型避免了锁竞争，具有更好的并发性能，而Array Map在需要频繁读取的场景下表现优异。文章还讨论了这些Map类型在不同系统环境中的适用场景，比如高效数据传输、动态数据存储和多核环境中的性能优化。通过这些分析，文章为设计eBPF程序提供了全面的理论基础和实践指导，有助于开发者在不同应用场景中合理选择和使用Map类型，从而提升程序的性能和效率。

这里举例说明一下BPF_MAP_TYPE_PERCPU_ARRAY的分析过程：

- **Per-CPU Array Map** 是一种数组，每个 CPU 都有自己独立的数组副本。每个数组中的元素存储一个键值对，且所有 CPU 的数组结构是相同的。

- 与普通的 Array Map 类似，**Per-CPU Array Map** 也需要在创建时指定最大大小，因为数组的大小在运行时是不可变的。
- **Per-CPU Array Map** 的实现相对简单，可以通过数组索引直接访问元素，因此插入和查找操作都非常高效。
- 适合**需要高并发读写且按索引访问**的场景，尤其是在多核系统中，因为每个 CPU 的数组操作都是独立的。
- 适合**在运行时已知最大键值对数量**的场景，因为它在创建时需要指定数组的最大大小。

1.per-cpu-array的创建源码：（/kernel/bpf/arraymap.c）

```
static int bpf_array_alloc_percpu(struct bpf_array *array)
{
    // 用于存储 per-cpu 分配的指针
    void __percpu *ptr;
    int i;

    // 遍历每个数组条目，为每个条目分配 per-cpu 的内存
    for (i = 0; i < array->map.max_entries; i++) {
        // 调用 bpf_map_alloc_percpu 为当前数组条目分配 per-cpu 内存
        ptr = bpf_map_alloc_percpu(&array->map, array->elem_size, 8,
                                   GFP_USER | __GFP_NOWARN);
        if (!ptr) {
            // 如果分配失败，释放之前已分配的 per-cpu 内存
            bpf_array_free_percpu(array);
            return -ENOMEM;  // 返回内存分配失败错误
        }
        // 将分配的 per-cpu 内存指针存储到数组的 pptrs（per-cpu 指针数组）中
        array->pptrs[i] = ptr;

        // 在分配过程中调用 cond_resched，以便让内核有机会调度其他任务
        cond_resched();
    }

    // 所有数组条目成功分配 per-cpu 内存后，返回 0，表示成功
    return 0;
}
```

该函数负责为 `BPF_MAP_TYPE_PERCPU_ARRAY` 类型的 BPF 数组的每个元素分配独立的 per-cpu 内存。循环遍历每个数组元素，调用 `bpf_map_alloc_percpu` 为每个元素分配 per-cpu 内存，将分配的 per-cpu 内存指针存储到 `array->pptrs` 中，方便后续使用。该函数的目标是确保 `BPF_MAP_TYPE_PERCPU_ARRAY` 的每个条目都拥有独立的 per-cpu 内存空间，以便在多 CPU 环境中高效处理数据。

2.per-cpu-array的查找源码：（/kernel/bpf/arraymap.c）

```
static void *percpu_array_map_lookup_elem(struct bpf_map *map, void *key)
{
    // 将 bpf_map 结构体转换为 bpf_array 结构体，以便访问数组特有的字段
    struct bpf_array *array = container_of(map, struct bpf_array, map);
    // 将 key 转换为 u32 类型，作为数组的索引
```

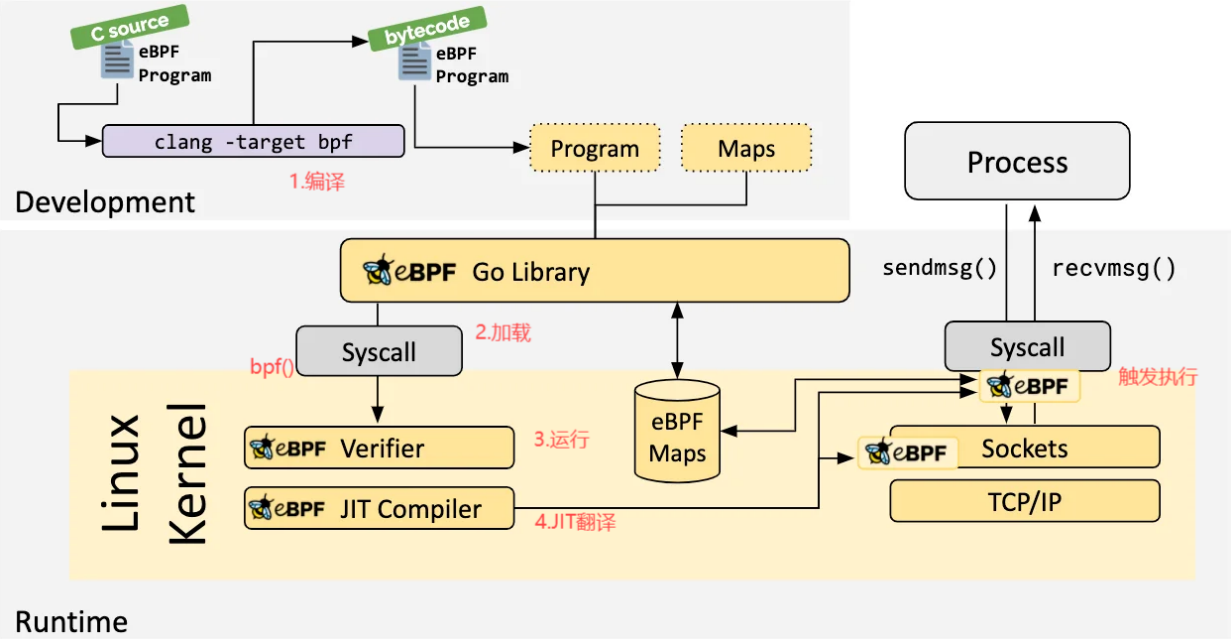
```
    u32 index = *(u32 *)key;

    // 检查索引是否超出数组的最大有效索引范围
    if (unlikely(index >= array->map.max_entries))
        return NULL;

    // 使用 this_cpu_ptr 宏获取当前 CPU 上的值
    // array->pptrs 存储每个索引的 per-cpu 指针
    // 通过 index & array->index_mask 确保索引在有效范围内
    return this_cpu_ptr(array->pptrs[index & array->index_mask]);
}
```

1.4深入探析 eBPF从程序编写到执行的全流程解析

在本项目中，我也对eBPF从程序编写到执行的全流程进行了详细的解析，并且输出了一个文档，由于文档较大，这里就不展示详细内容了，在项目仓库中的docs文件夹下。这里简要介绍：从以下四个图中所标注的方面来分析eBPF的执行原理：



这张图是eBPF官方给出的图，我会从图中标注的四个重点步骤来简要说明一下：

> 1.编译：
>
> - eBPF 程序以 C 语言编写，使用 `clang -target bpf` 编译器将 C 源代码编译为 eBPF 字节码。这个字节码是内核可以理解并执行的代码格式。
> - 编译之后，生成的 eBPF 程序字节码和与之相关的 eBPF 映射（Maps）可以通过用户态程序进行管理。
>
> 2.加载：
>
> - 编译生成的 eBPF 程序会通过 `libbpf` 库加载到 Linux 内核中。这个步骤涉及使用 `bpf()` 系统调用将 eBPF 程序提交给内核进行处理。
> - 同时，eBPF Maps（存储 eBPF 程序运行时数据的结构）也会在用户态与内核之间进行交互。
>
> 3.验证：

> - 在程序加载后，eBPF 验证器会检查字节码，确保程序的安全性和合法性，防止 eBPF 程序破坏内核的稳定性。验证通过后，eBPF 程序通过 eBPF JIT（Just-In-Time）编译器进一步优化，将字节码转换为机器码以提升执行效率。
>
> 4.JIT：
>
> - 通过 JIT，eBPF 字节码被直接编译成底层的机器代码，程序可以直接在 CPU 上执行，无需经过解释过程。

在原定方案上，我基本完成了相关测试和原理层面上的详细分析。但是还有一些测试并不完善，包括：内核版本之间的Map性能比较、不同挂载点类型的差异和特点。

2.遇到的问题以及解决方案：

1.问题1：编写 eBPF 程序时编译报错

- **场景**：在编写 eBPF 程序时，使用 clang 编译时出现错误提示，例如“不支持的 eBPF 指令”或“程序太复杂”。
- **原因分析**：eBPF 程序在编写时需要遵守严格的验证器规则，eBPF 不支持一些复杂的 C 语言特性（如递归、动态内存分配等）。此外，编译时可能会因为程序过于复杂导致验证失败。
- **解决方案：**
  1. **简化代码**：避免使用递归或复杂的循环，尽量将逻辑拆分为简单的代码片段。
  2. **检查 BPF Helper 函数**：确保所使用的 Helper 函数是 eBPF 所支持的函数集。
  3. **使用 clang 的 `-O2` 优化选项**：这个优化级别有助于生成通过验证器检查的 eBPF 字节码。
  4. **查看错误日志**：编译时启用日志选项 `-g` 和 `-O2`，通过 `bpftool` 等工具获取详细的错误日志。

2. 问题2：使用 Per-CPU Map 时无法读取 map 中的数据

- **场景**：使用 `BPF_MAP_TYPE_PERCPU_ARRAY` 或 `BPF_MAP_TYPE_PERCPU_HASH` 存储数据时，尝试在用户空间读取数据，发现读取到的值不正确或为空。
- **原因分析**：Per-CPU Map 中的数据是针对每个 CPU 单独存储的，读取时需要汇总所有 CPU 的数据。如果只读取某一个 CPU 的数据，而忽略了其他 CPU 的数据，就可能导致数据不完整或读取失败。
- **解决方案：**
  1. **汇总所有 CPU 数据**：在读取数据时，通过 `bpf_map_lookup_elem` 遍历每个 CPU 并读取各自的数据，然后汇总到一起。
  2. **检查当前 CPU 上的数据**：如果只想读取某个 CPU 上的数据，确保调用了合适的 Helper 函数，并在正确的 CPU 上进行数据操作。
  3. **用户空间读取方式**：使用 `bpf_map_lookup_percpu_elem` 等函数在用户空间正确获取 Per-CPU Map 中的数据。

3. 问题3：查找 eBPF 原理资料和论文时，有些内容自己不太明白

- **场景**：在阅读 eBPF 相关文献或博客时，对某些复杂的技术细节或底层原理（如 JIT 编译、验证器工作原理等）感到困惑。
- **原因分析**：eBPF 涉及的领域广泛，包括内核、网络、虚拟机等，部分技术细节理解起来较为复杂，尤其是涉及到内核机制、性能优化等底层知识时。

- **解决方案：**

    1. **循序渐进地学习**：从基础开始，逐步深入，先掌握 eBPF 的基本工作机制（如 Map、Helper 函数、加载机制等），再深入学习 JIT 编译、验证器等较复杂的内容。
    2. **多参考官方文档和社区资源**：eBPF 社区提供了大量的学习资料，可以通过 Linux Documentation 或 bcc、libbpf等参考官方文档和示例代码。
    3. **加入社区讨论**：参与 eBPF 社区的讨论，向有经验的开发者请教，可以加快对复杂概念的理解。
    4. **查阅论文和技术报告**：一些研究论文或报告中对 eBPF 的底层实现有详细的解释，可以结合实战和代码理解其工作原理。

4. 问题4：eBPF 程序验证器不通过

- **场景**：在运行 eBPF 程序时，加载阶段遇到验证器失败，导致程序无法执行。
- **原因分析**：eBPF 验证器用于确保 eBPF 程序的安全性和正确性，如果程序使用了内核不允许的操作（如越界访问、非法跳转等），验证器将拒绝加载该程序。
- **解决方案：**

    1. **减少程序复杂性**：将 eBPF 程序的复杂度降到最小，确保每个函数内的逻辑尽量简单。
    2. **查看错误原因**：通过 bpftool 或 dmesg 查看详细的验证器错误日志，定位问题所在。
    3. **分步调试**：将 eBPF 程序分解成多个小段，每段单独调试，找到验证器不通过的具体原因。

5. 问题5：用户空间与内核空间的数据同步问题

- **场景**：在 eBPF 程序中通过 map 向用户空间传递数据时，发现数据同步存在延迟或不一致的问题。
- **原因分析**：eBPF Map 的数据同步不是实时的，有时需要手动触发同步或使用特定方式来确保数据的一致性。
- **解决方案：**

    1. **使用 BPF Ring Buffer**：使用 `BPF_MAP_TYPE_RINGBUF` 代替传统的 Map，可以实现内核与用户空间之间的高效数据传输。
    2. **使用 Perf Buffer**：通过 `BPF_MAP_TYPE_PERF_EVENT_ARRAY` 实现更高效的事件数据传输，避免延迟问题。
    3. **手动同步**：在用户空间定期轮询数据或通过触发事件确保数据及时同步。

3.后续工作安排：

由于本次测试还有一些测试工作没有完善，因此后续会继续推进该项目，具体包括：

在后续工作中，将进一步深入测试eBPF的性能表现，重点聚焦在不同内核版本以及挂载点上的性能对比。首先，将对不同Linux内核版本进行详细的性能比较。计划选取几个具有代表性的内核版本，分别进行eBPF程序的加载与运行测试。通过对比分析不同内核版本下eBPF程序的执行时间、CPU使用率、内存消耗等核心指标，评估其对各种Map类型（如Hash、Array、Per-CPU Map）的性能影响。同时，针对JIT编译的效率差异也将进行测试，尤其是在高并发场景中不同内核版本对编译效率的影响。此外，还会比较各版本的验证器表现，分析验证器在不同版本中的严格性以及处理时间的差异，最终形成详细的性能对比报告并提出优化建议。

在此基础上，还将对eBPF程序在不同挂载点上的性能表现进行测试，评估挂载点选择对程序性能的影响。测试将覆盖多种挂载点类型，包括TC（Traffic Control）、XDP（eXpress Data Path）、Tracepoints和Kprobes等。在每个挂载点上，将运行相同的eBPF程序，并重点分析数据包处理延迟、吞吐量、CPU与内存的使用情况。通过高负载场景测试，将明确不同挂载点在处理能力上的差异，并结合它们各自的特性，提出适用于不同应用场景的挂载点选择建议。

最后，基于测试结果，会提出针对性优化方案，尤其是在不同内核版本与挂载点组合中找到性能最优的配置。通过持续测试和反馈，将确保eBPF程序的性能随着新的内核版本或硬件配置不断得到优化和改进。