ExtFUSE中lookup优化的大致过程的内核代码梳理：

Extfuse的ebpf框架分为4个部分：

- linux内核增加注册extfuse的ebpf的程序类型；
- 在FUSE的内核中增加ebfp的挂载点以及相应的钩子函数，并增加辅助函数；
- 设计相应的ebpf挂载函数：HANDLER(FUSE_xxx)；
- 在用户态文件系统建立与内核共同使用的ebpf map，并在相关的元数据操作中维护ebpf map。

1、增加extfuse ebpf程序类

linux内核的地址https://github.com/extfuse/linux

在bpf_types.h中定义BPF_PROG_TYPE(BPF_PROG_TYPE_EXTFUSE, extfuse) 类型

在bpf.h中增加bpf_prog_type，BPF_PROG_TYPE_EXTFUSE

在bpf_load.c中增加对程序类型extfuse的内容加载，extfuse的程序加载使用prog array的索引方式。

```
struct bpf_map_def SEC("maps") handlers = {
    .type = BPF_MAP_TYPE_PROG_ARRAY,
    .key_size = sizeof(u32),
    .value_size = sizeof(u32),
    .max_entries = FUSE_OPS_COUNT << 1,
};
```

通过 `BPF_MAP_TYPE_PROG_ARRAY`，可以将多个 eBPF 程序存储在一个数组中，每个程序都有一个唯一的索引。使用 `BPF_MAP_TYPE_PROG_ARRAY` 类型的 BPF Map，可以轻松管理和访问多个 eBPF 程序。

加载的规则如下，其中根据sec(“extfuse")来搜索这个类型的ebpf程序，section字符串为“extfuse\1“就加载到索引为1的ebpf程序中，section字符串为“extfuse\2“加载到索引为2的ebpf程序中。在extfuse.c文件中说明了这种定义的方式。

```
#define HANDLER(F) SEC("extfuse/"__stringify(F)) int bpf_func_##F
```

int SEC("extfuse") fuse_xdp_main_handler(void *ctx) 加载的主ebpf程序

```
int SEC("extfuse") fuse_xdp_main_handler(void *ctx)
{
    struct extfuse_req *args = (struct extfuse_req *)ctx;
    int opcode = (int)args->in.h.opcode;
    PRINTK("Opcode %d\n", opcode);
                bpf_tail_call(ctx, &handlers, opcode);
                return UPCALL;
}
```

HANDLER(FUSE_LOOKUP)(void *ctx) 加载的分支ebpf程序，其中#define FUSE_LOOKUP 1，其中fuse_xdp_main_handler通过extfuse的section中的数字部分op来确定加载那个ebpf的分支程序。

2、FUSE内核中挂载点和辅助函数

```
static void process_init_reply(struct fuse_conn *fc, struct fuse_req *req)
//处理FUSE文件系统的初始化回复
{
    …
    if (arg->flags & FUSE_FS_EXTFUSE)   //FUSE_FS_EXTFUSE标志确定是否加载eBPF程序
    extfuse_load_prog(fc, arg->extfuse_prog_fd); //加载ebpf
    …
}
int extfuse_request_send(struct fuse_conn *fc, struct fuse_req *req)    //发送扩展的FUSE请求
{
    …
    if (data) {
        struct extfuse_req ereq;
        fuse_to_extfuse_req(req, &ereq);    //将fuse_req结构体转换为extfuse_req结构体
        ret = extfuse_run_prog(data->prog, &ereq); //执行ebpf程序并执行
        if (ret != -ENOSYS) {
            extfuse_to_fuse_req(&ereq, req);
            req->out.h.error = (int)ret;
            ret = 0;
        }
    }
    …
}
```

建立程序需要运行的ebpf调用过程中的辅助函数

```
BPF_CALL_4(bpf_extfuse_read_args, void *, src, u32, type, void *, dst, size_t, size)
BPF_CALL_4(bpf_extfuse_write_args, void *, dst, u32, type, const void *, src, u32, size)
//用于在BPF程序中通过函数ID查找相应的函数原型
static const struct bpf_func_proto *
bpf_extfuse_func_proto(enum bpf_func_id func_id, const struct bpf_prog *prog)
{
    switch (func_id) {
    case BPF_FUNC_extfuse_read_args:
        return &bpf_extfuse_read_args_proto;
    case BPF_FUNC_extfuse_write_args:
        return &bpf_extfuse_write_args_proto;
    case BPF_FUNC_map_lookup_elem:
        return &bpf_map_lookup_elem_proto;
    case BPF_FUNC_map_update_elem:
        return &bpf_map_update_elem_proto;
    case BPF_FUNC_map_delete_elem:
        return &bpf_map_delete_elem_proto;
    case BPF_FUNC_tail_call:
        return &bpf_tail_call_proto;
    case BPF_FUNC_trace_printk:
        return bpf_get_trace_printk_proto();
    default:
```

```
        return NULL;
    }
}
```

向ebpf注册相应的辅助操作，用于存储与BPF验证器相关的操作函数

```
const struct bpf_verifier_ops extfuse_verifier_ops = {
    .get_func_proto = bpf_extfuse_func_proto,    //用于获取BPF函数的函数原型
    .is_valid_access = bpf_extfuse_is_valid_access, //用于判断BPF程序是否可以访问指定的资源
};
```

3、ebpf用户态程序

定义inode cache，具体用于元数据的查询、插入操作中

```
struct bpf_map_def SEC("maps") entry_map = {
    .type = BPF_MAP_TYPE_HASH, // simple hash list
    .key_size = sizeof(lookup_entry_key_t),
    .value_size = sizeof(lookup_entry_val_t),
    .max_entries = MAX_ENTRIES,
    .map_flags = BPF_F_NO_PREALLOC,
};
```

1. `lookup_entry_key_t` 结构体：
   - `uint64_t nodeid`：父节点的 ID。
   - `char name[NAME_MAX]`：节点名称。
2. `lookup_entry_val_t` 结构体：
   - `uint32_t stale`：标记该条目是否过时。
   - `uint64_t nlookup`：引用计数。
   - `uint64_t nodeid`：子节点的 ID。
   - `uint64_t generation`：节点的代数。
   - `uint64_t entry_valid`：条目的有效性。
   - `uint32_t entry_valid_nsec`：条目有效性的纳秒部分。

定义inode attr cache，具体用于元数据信息的更新、删除、判断是否有效等操作中。

```
struct bpf_map_def SEC("maps") attr_map = {
    .type = BPF_MAP_TYPE_HASH, // simple hash list
    .key_size = sizeof(lookup_attr_key_t),
    .value_size = sizeof(lookup_attr_val_t),
    .max_entries = MAX_ENTRIES,
    .map_flags = BPF_F_NO_PREALLOC,
};
```

1. `lookup_attr_key_t` 结构体：
   - `uint64_t nodeid`：节点的 ID。
2. `lookup_attr_val_t` 结构体：
   - `uint32_t stale`：标记该属性是否过时。
   - `struct fuse_attr_out out`：结构体包含节点属性的详细信息。

定义 ebpf钩子函数

```
HANDLER(FUSE_LOOKUP)(void *ctx)
{
    …
    lookup_entry_val_t *entry = bpf_map_lookup_elem(&entry_map, &key); //在map中查找inode的emtry条目，找到的话，更新输出结果，直接返回
    if (!entry || entry->stale) {
        if (entry && entry->stale)  //emtry条目过期
            PRINTK("LOOKUP: STALE key name: %s nodeid: 0x%llx\n",key.name,key.nodeid);
        else                        //不存在emtry条目
            PRINTK("LOOKUP: No entry for node %s\n", key.name);
        return UPCALL;
    }
    PRINTK("LOOKUP(0x%llx, %s): nlookup %lld\n",
    key.nodeid, key.name, entry->nlookup);
    /* prepare output */
    struct fuse_entry_out out;
    uint64_t nodeid = entry->nodeid;
    /* negative entries have no attr */
    if (!nodeid) {
        create_lookup_entry(&out, entry, NULL);
    } else {
        lookup_attr_val_t *attr = bpf_map_lookup_elem(&attr_map, &nodeid); //查看inodeid是否在map中
        if (!attr || attr->stale) {
            if (attr && attr->stale)
                PRINTK("LOOKUP: STALE attr for node: 0x%llx\n", nodeid);
            else {
                PRINTK("LOOKUP: No attr for node 0x%llx\n", nodeid);
                return UPCALL;
            }
        }
        PRINTK("LOOKUP nodeid 0x%llx attr ino: 0x%llx\n",
        entry->nodeid, attr->out.attr.ino);
        create_lookup_entry(&out, entry, &attr->out);
    }
    //使用bpf的辅助函数把参数写到返回请求
    /* populate output */
    ret = bpf_extfuse_write_args(ctx, OUT_PARAM_0, &out, sizeof(out));
    if (ret) {
        PRINTK("LOOKUP: Failed to write param 0: %d!\n", ret);
        return UPCALL; //不满足条件，继续后面的请求
    }
    //完成请求返回
    return RETURN;
}
```

extfuse.c中还包括HANDLER(FUSE_GETATTR)、HANDLER(FUSE_READ)、HANDLER(FUSE_WRITE)、HANDLER(FUSE_SETATTR)、HANDLER(FUSE_GETXATTR)、HANDLER(FUSE_FLUSH)、HANDLER(FUSE_RENAME)、HANDLER(FUSE_RMDIR)、HANDLER(FUSE_UNLINK)这几个操作的函数框架。

其中还额外包括一个函数：

```
static int remove(void *ctx, int param, char *op, lookup_entry_key_t *key)
```

用于删除引用计数为0的文件元数据。

4、在用户态文件系统中更新map

在文件系统初始化时加载并初始化ebpf程序

```
static void stackfs_ll_init(void *userdata, struct fuse_conn_info *conn)
    lo->ebpf_ctxt = ebpf_init("/tmp/extfuse.o");
```

插入inode到eBPF的map中，并更新缓存

```
int create_entry(fuse_req_t req, struct lo_inode* pinode,const char *name,
const char *cpath,struct fuse_entry_param *e, const char *op, int time)
…
#ifdef ENABLE_EXTFUSE_LOOKUP
    res = lookup_fetch(get_lo_data(req)->ebpf_ctxt, inode->pino,inode-
>name);
    if (res && !errno) {
        INFO("[%d] \t Fetched %s nlookup: %ld inode->nlookup: %ld\n",
        gettid(), res < 0 ? "stale" : "", res, inode->nlookup);
        if (res < 0)
            res = -res;
        inode->nlookup = res + 1;
    }
    //插入inode到map中去
    res = lookup_insert(get_lo_data(req)->ebpf_ctxt, inode->pino,
    inode->name, inode->nlookup, e);
    if (res)
        ERROR("[%d] \t %s new node %s id: %p: %s\n",
    gettid(), op, cpath, inode, strerror(errno));
#endif
…
    int lookup_insert(ebpf_context_t *ctxt, uint64_t parent_ino,
    const char *name, uint64_t nlookup, struct fuse_entry_param *e)
    INFO("[%d] \t Inserting node name %s (%ju) parent 0x%lx nlookup; %lu\n",
    gettid(), name, strlen(name), parent_ino, nlookup);
    // update entry，调用ebpf map接口更新inode cache map，0表示inode cache map，
1 表示attr cache map
    int overwrite = 1; //XXX overwiting to update any negative entires
    ret = ebpf_data_update(ctxt, (void *)&key, (void *)&entry, 0,
overwrite);
    if (ret)
        ERROR("[%d] \t Failed to insert %s (%ju) parent 0x%lx count %ju:
%s\n",
    gettid(), name, strlen(name), parent_ino, num_entries, strerror(errno));
```

其中涉及了一些操作inode的函数，create_entry，delete_node，forget_inode，stackfs_ll_rename。