



第三届 eBPF开发者大会

www.ebpftravel.com

迈向eBPF并发领域的内存模型

范佳明·南京大学

中国·西安

目录

- 问题背景
- 什么是 BPF 内存模型
- BPF 指令的内存语义
- 与 LLVM 的集成
- 未来方向

0. 问题背景

在很多开发者心中，eBPF 是不是一种简单的、单线程的沙箱系统？

但你有没有想过：多个 BPF 程序是否可能同时访问共享内存？

0. 问题背景

在很多开发者心中，eBPF 是不是一种简单的、单线程的沙箱系统？

但你有没有想过：多个 eBPF 程序是否可能同时访问共享内存？

答案是：当然可以，甚至已经发生了。

0. 问题背景

一个 race condition 的例子，如果下面的代码在多个 CPU 上同时运行

```
// cpu 1
u64 *v = bpf_map_lookup_elem(&my_map, &(u32){0});
if (v) (*v) += 1

// cpu 2
u64 *v = bpf_map_lookup_elem(&my_map, &(u32){0});
if (v) bpf_printk("v: %llu\n", *v);
```

bpf_printk 的结果应该是什么：??

0. 问题背景

一个 race condition 的例子，如果下面的代码在多个 CPU 上同时运行

```
// cpu 1
u64 *v = bpf_map_lookup_elem(&my_map, &(u32){0});
if (v) (*v) += 1

// cpu 2
u64 *v = bpf_map_lookup_elem(&my_map, &(u32){0});
if (v) bpf_printk("v: %llu\n", *v);
```

bpf_printk 的结果应该是什么：**取决于内存模型**

0. 问题背景

什么是内存模型？

A memory model defines outcomes of concurrent accesses

-- Paul E. McKenney

0. 问题背景

举例来说，对于同一内存 ($a = 0$)，一读 (`read(a)`) 一写 ($a = 1$) 的情况，不同的内存模型如何定义结果？

- C/C++ 内存模型：未定义行为 (UB)
 - “Any data race results in undefined behavior.” — C++11 standard §1.10
 - 在理论模型中，你甚至可能观察到 $a == 42$ ，即 OOTA (Out-of-Thin-Air) 行为

0. 问题背景

举例来说，对于同一内存 ($a = 0$)，一读 (`read(a)`) 一写 ($a = 1$) 的情况，不同的内存模型如何定义结果？

- C/C++ 内存模型：未定义行为 (UB)
 - “Any data race results in undefined behavior.” — C++11 standard §1.10
 - 在理论模型中，你甚至可能观察到 $a == 42$ ，即 OOTA (Out-of-Thin-Air) 行为
- Linux Kernel Model：要么是新值 1，要么是旧值 0，但不会是凭空出现的值 (OOTA)
 - 内核文档 `memory-barriers.txt` → section: "All memory-reference instructions"

0. 问题背景

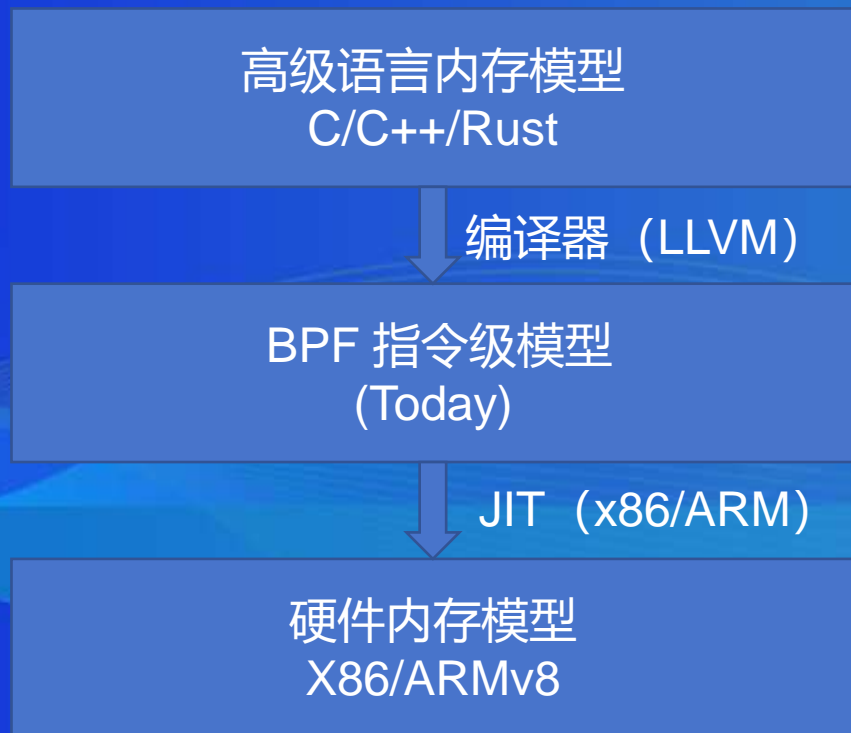
那么，eBPF 程序的内存模型是什么呢？

实际上，eBPF 的内存模型一直没有被提出，BPF 只是 instruction set，却没有定义自己的 memory model —— 直到 2023 年，Paul E. McKenney 提出了《Instruction-Level BPF Memory Model》[1]

[1] <https://docs.google.com/document/d/1TaSEfWfLnRUi5KqkavUQyL2tThJXYWHS15qcbxlsFb0>

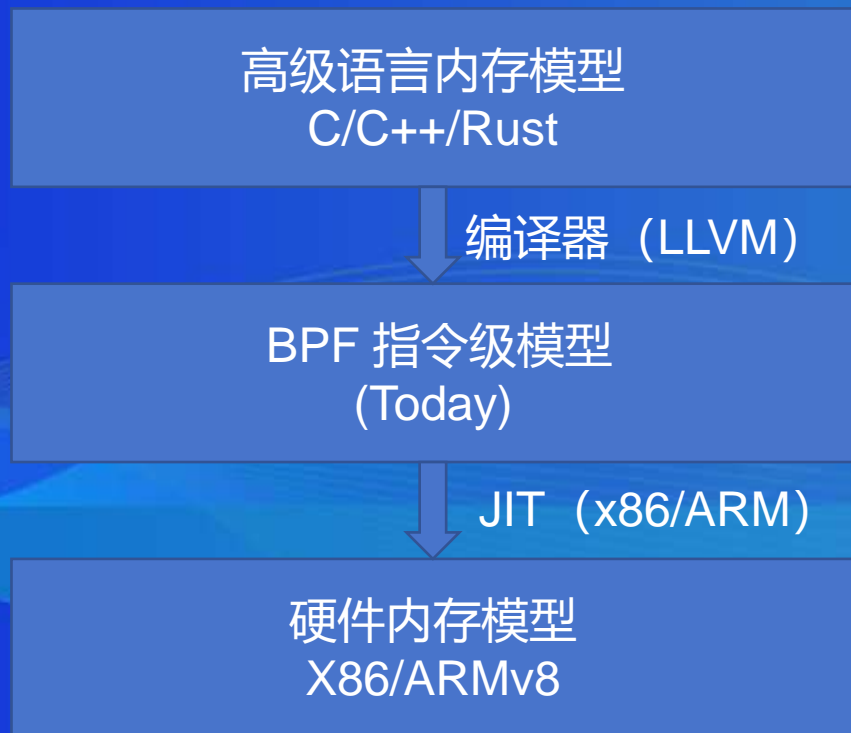
1. 什么是 BPF 内存模型

三层并存的内存模型结构



1. 什么是 BPF 内存模型

三层并存的内存模型结构



BPF 内存模型就处于将高级语言转接到硬件指令的转接层

举例：BPF 程序中 `store_release()` → *LLVM* → BPF → *JIT* → ARMv8

如果 BPF 无法表达 store 是 release 的，那么下层 jit 可能不会插入合适的 memory-barrier，导致程序出现诡异的错误！

1. 什么是 BPF 内存模型

为了理解 BPF 内存模型，我们到底要理解什么

我们说了这么多，回到问题本身：如果要理解 BPF 的 memory model，我们应该去理解什么？

是 helper 函数？是 Verifier？是整个 eBPF 程序

——都不是。

真正构成这个模型基础的，是一条条 BPF 指令本身。换句话说，我们要去理解每一类指令的顺序语义

2. BPF 指令的内存语义

不是所有指令都影响 memory order, 下面 4 类指令带有顺序语义

- 原子指令 (Atomic Instructions)
- 跳转指令 (Jump Instructions)
- 加载指令 (Load instructions)
- 内存引用指令 (Memory-Reference Instructions)

2. BPF 指令的内存语义

原子指令 (Atomic Instructions)

- BPF_XCHG, BPF_CMPXCHG
 - 这两条指令具备全序 (full ordering) 属性, 也就是说, 它自带双向屏障, 确保不会被重排序。
 - 与 Linux 内核中的 `atomic_xchg()` 和 `atomic_cmpxchg()` 一致
 - 等价于下面的代码:
 - `smp_mb();`
 - `atomic_cmpxchg_relaxed(ptr, old, new);`
 - `smp_mb();`

2. BPF 指令的内存语义

原子指令 (Atomic Instructions)

- BPF_ADD / BPF_OR / BPF_AND / BPF_XOR
 - 这些指令只保证操作是原子的，没有任何顺序保证，可被CPU/JIT优化中被任意重排
 - 对应内核的 `atomic_add()` / `atomic_or()` / `atomic_and()` / `atomic_xor()`
 - 很多开发者以为这些操作是“绝对安全的”，其实是错的！它们是 race-safe，但不是 order-safe。

那么如果确实需要 order-safe 的上述操作呢？

2. BPF 指令的内存语义

原子指令 (Atomic Instructions)

- 用 BPF_FETCH 修饰的 BPF_ADD / BPF_OR / BPF_AND / BPF_XOR
 - 如果这些操作使用 BPF_FETCH 修饰, 那么它们就变成全序了。即: “读 + 操作 + 写” 作为单个原子单元, 且两侧都有 memory barrier 行为

2. BPF 指令的内存语义

原子指令 (Atomic Instructions)

内核的 JIT compiler 遵循这种语义吗?

看看源码, 以 ARM64 为例

2. BPF 指令的内存语义

```
switch (insn->imm) {
    /* lock *(u32/u64 *) (dst_reg + off) <op>= src_reg */
    case BPF_ADD:
        emit(A64_STADD(isdw, reg, src), ctx);
        break;
    case BPF_AND:
        emit(A64_MVN(isdw, tmp2, src), ctx);
        emit(A64_STCLR(isdw, reg, tmp2), ctx);
        break;
    case BPF_OR:
        emit(A64_STSET(isdw, reg, src), ctx);
        break;
    case BPF_XOR:
        emit(A64_STEOR(isdw, reg, src), ctx);
        break;
    /* src_reg = atomic_fetch_<op>(dst_reg + off, src_reg) */
    case BPF_ADD | BPF_FETCH:
        emit(A64_LDADDAL(isdw, src, reg, src), ctx);
        break;
    case BPF_AND | BPF_FETCH:
        emit(A64_MVN(isdw, tmp2, src), ctx);
        emit(A64_LDCLRAL(isdw, src, reg, tmp2), ctx);
        break;
    case BPF_OR | BPF_FETCH:
        emit(A64_LDSETAL(isdw, src, reg, src), ctx);
        break;
    case BPF_XOR | BPF_FETCH:
        emit(A64_LDEORAL(isdw, src, reg, src), ctx);
        break;
}
```

2. BPF 指令的内存语义

```
switch (insn->imm) {  
    /* lock *(u32/u64 *) (dst_reg + off) <op>= src_reg */  
    case BPF_ADD:  
        emit(A64_STADD(isdw, reg, src), ctx);  
        break;  
    case BPF_AND:  
        emit(A64_MVN(isdw, tmp2, src), ctx);  
        emit(A64_STCLR(isdw, reg, tmp2), ctx);  
        break;  
    case BPF_OR:  
        emit(A64_STSET(isdw, reg, src), ctx);  
        break;  
    case BPF_XOR:  
        emit(A64_STEOR(isdw, reg, src), ctx);  
        break;  
    /* src_reg = atomic_fetch_<op>(dst_reg + off, src_reg) */  
    case BPF_ADD | BPF_FETCH:  
        emit(A64_LDADDAL(isdw, src, reg, src), ctx);  
        break;  
    case BPF_AND | BPF_FETCH:  
        emit(A64_MVN(isdw, tmp2, src), ctx);  
        emit(A64_LDCLRAL(isdw, src, reg, tmp2), ctx);  
        break;  
    case BPF_OR | BPF_FETCH:  
        emit(A64_LDSETAL(isdw, src, reg, src), ctx);  
        break;  
    case BPF_XOR | BPF_FETCH:  
        emit(A64_LDEORAL(isdw, src, reg, src), ctx);  
        break;  
}
```

如果你写的是 BPF_ADD, 它被翻译成 STADD —— 一个无 barrier 的原子指令。

但如果你写的是 BPF_FETCH | ADD, 它就变成了 LDADDAL —— 自带 acquire+release 语义。

所以, FETCH 修饰符不仅改变了返回值类型, 更关键地, 它改变了内存顺序语义。

2. BPF 指令的内存语义

跳转指令 (Jump Instructions)

- 无条件跳转指令 (BPF_JA / BPF_CALL / BPF_EXIT) 不提供任何内存序保证
- 有条件跳转指令：
 - BPF_JEQ, BPF_JNE, BPF_JGT, BPF_JGE, BPF_JLT, BPF_JLE, BPF_JSET, BPF_JSGT, BPF_JSGE, BPF_JSLT, BPF_JSLE
 - 上述指令在某些条件下可以建立控制依赖 (control dependency) , 从而提供弱序 (weak order)

2. BPF 指令的内存语义

跳转指令 (Jump Instructions)

- “某些条件” 是指?
 - 分支条件依赖于先前的 load 指令
 - 分支内有 store 指令
 - 该 store 在 “控制流汇合点之前” 执行
 - 举例: `if (*flag) { *ptr = 1; }`

2. BPF 指令的内存语义

跳转指令 (Jump Instructions)

- 什么是 Weak Order ?
 - 即不会提供传递性 (non-cumulativity) , 它只能作为一个局部顺序提示, 让后续 store 不早于前面的 load
 - 根据 memory-barriers.txt 的说法: control dependencies do not accumulate across CPUs.

2. BPF 指令的内存语义

跳转指令 (Jump Instructions)

- 什么是 Weak Order ? 一个更直观的例子

```
CPU 0:
  if (*x)
    *y = 1;    // x → y: control dependency

CPU 1:
  if (*y)
    *z = 1;    // y → z: control dependency

CPU 2:
  if(*z)
    assert(*x==1)    // *x might be 0 !
```

当某个 CPU 把 *x 改为 1 后

即使 CPU2 观察到 *z 的值为 1 时, *x 也不一定为 1, 即不具备传递性

2. BPF 指令的内存语义

加载指令（ Load Instructions ）

- BPF 的 Load 指令本身不提供任何 memory ordering，它们不是 acquire-load，也不会插入 barrier。
- 但如果 load 的结果被用于构造后续指令的地址或值，就可以建立一种地址依赖（address-dependency）或者数据依赖（data-dependency）

2. BPF 指令的内存语义

加载指令 (Load Instructions)

- BPF 的 Load 指令本身不提供任何 memory ordering, 它们不是 acquire-load, 也不会插入 barrier。
- 但如果 load 的结果被用于构造后续指令的地址或值, 就可以建立一种地址依赖 (address-dependency) 或者数据依赖 (data-dependency)
- 举例而言:
 - 地址依赖: $r1 = x; r2 = r1 + 1$
 - 数据依赖: $r1 = *x; *y = r1 + 1$

2. BPF 指令的内存语义

加载指令 (Load Instructions)

- BPF 的 Load 指令本身不提供任何 memory ordering, 它们不是 acquire-load, 也不会插入 barrier。
- 但如果 load 的结果被用于构造后续指令的地址或值, 就可以建立一种地址依赖 (address-dependency) 或者数据依赖 (data-dependency)
- 地址依赖和数据依赖都是 weak order, 与控制依赖类似

2. BPF 指令的内存语义

内存引用指令（Memory-Reference Instructions）

- 对**同一个**地址的 load/store 具有全局顺序一致性
- 这就是我们熟悉的现代多核架构中最基础的保证之一：Single-variable coherence order（单变量内存一致性）

2. BPF 指令的内存语义

内存引用指令 (Memory-Reference Instructions)

- 对**同一个**地址的 load/store 具有全局顺序一致性
- 这就是我们熟悉的现代多核架构中最基础的保证之一：Single-variable coherence order (单变量内存一致性)
- 举例：

$X = 1$

$X = 2$

所有 CPU 都看到 $X = 1$ 、 2 的顺序

3. 与 LLVM 的集成

这些模型和我写的代码有什么关系？我只是想原子地加个值而已啊！

这一节我们从程序员的视角来看一看，如果我想在代码中完成某些内存序的指定，我该如何做？有哪些 API 可用？

3. 与 LLVM 的集成

与内存序有关的 API: `__atomic_*`() 和 `__sync_*`()

- `__atomic_*`() 和 `__sync_*`() 是 GCC 最早定义的内建函数 (built-ins) ,
 - 但 Clang/LLVM 也兼容支持了这些 API
 - 本PPT后续讨论的行为都基于 Clang/LLVM 对这些 built-in 的支持与实现.
-
- 注: 目前 Clang/LLVM 对 BPF 后端对原子操作支持有限。现在还有很多 API 并不能够完全翻译为 BPF 指令, 尚未支持的API我们需要手动实现

3. 与 LLVM 的集成

LLVM 目前支持有限

```
void test_mm1(unsigned long long *val) {  
    __atomic_fetch_add(val, 1, __ATOMIC_SEQ_CST);  
}
```

```
> clang -O2 -target bpf -c test.c -o test.bpf.o
```

成功编译通过

3. 与 LLVM 的集成

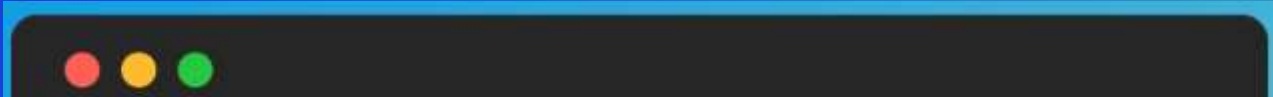
LLVM 目前支持有限

```
void test_mm2(unsigned long long *val) {  
    __atomic_store_n(val, 0, __ATOMIC_SEQ_CST);  
}
```

但是 __atomic_store_n 则尚未支持

3. 与 LLVM 的集成

LLVM 目前支持有限



```
> clang -O2 -target bpf -c test.c -o test.bpf.o
fatal error: error in backend: Cannot select: 0x55bd86cb60a0: ch = AtomicStore<(store seq_cst (s64) into %ir.0
)> 0x55bd86c5d638, Constant:i64<0>, 0x55bd86cb6180
0x55bd86cb6110: i64 = Constant<0>
0x55bd86cb6180: i64,ch = CopyFromReg 0x55bd86c5d638, Register:i64 %0
0x55bd86cb61f0: i64 = Register %0
In function: test_mm2
PLEASE submit a bug report to https://github.com/llvm/llvm-project/issues/ and include the crash backtrace, preprocessed source, and associated run script.
Stack dump:
0. Program arguments: clang -O2 -target bpf -c test.c -o test.bpf.o
1. <eof> parser at end of file
2. Code generation
3. Running pass 'Function Pass Manager' on module 'test.c'.
4. Running pass 'BPF DAG->DAG Pattern Instruction Selection' on function '@test_mm2'
```

所以后续有很多 API 目前需要我们自己实现

3. 与 LLVM 的集成

目前 LLVM 已经提供哪些 API

- `__atomic_exchange_n / __atomic_exchange`
- `__atomic_compare_exchange_n / __atomic_compare_exchange`
- `__atomic_{add | sub | and | or}_fetch / __atomic_fetch_{add | sub | and | or}`

他们的含义从字面上就能推断，而且底层映射的 BPF 指令也很直观

例如 `__atomic_exchange` 底层映射为 `BPF_XCHG`

3. 与 LLVM 的集成

目前 LLVM 没有提供哪些 API

- 全屏障: `bpf_full_barrier()`
- `__atomic_store_n` / `__atomic_store`
- `__atomic_load_n` / `__atomic_load`
- `__atomic_test_and_set`
- `__atomic_test_clear`
- `__atomic_thread_fence`

如果我们确实需要在 eBPF 中使用这些 API 呢? 自己造一个!

3. 与 LLVM 的集成

GCC 提供了哪些内存序?

- `__ATOMIC_RELAXED`
- `__ATOMIC_ACQUIRE`
- `__ATOMIC_CONSUME`
- `__ATOMIC_RELEASE`
- `__ATOMIC_ACQ_REL`
- `__ATOMIC_SEQ_CST`

- LLVM 为了兼容当然也提供了这些内存序

3. 与 LLVM 的集成

如何与 BPF 内存序对应?

- `__ATOMIC_RELAXED`: Relaxed ordering
- `__ATOMIC_ACQUIRE`
- `__ATOMIC_CONSUME`
- `__ATOMIC_RELEASE`
- `__ATOMIC_ACQ_REL`
- `__ATOMIC_SEQ_CST`

Full barrier ordering

目前 BPF 并不支持细粒度的 memory ordering 区分 (acquire/release)
所以 Clang 在编译 BPF 时, 干脆把所有 非-relaxed 的 memory order, 都映射为 full barrier

⚡ 等到 BPF 有了 acquire-release 语义后可能此处映射会再被修改

3. 与 LLVM 的集成

如何实现全屏障: `bpf_full_barrier()` ?

- BPF 中没有专门的全屏障指令, 但可以通过特定的 “原子指令模式” 来模拟出 full memory barrier 的效果:
 - `__sync_fetch_and_add(val, 0);`
 - 对任意值 +0, 配合 fetch 的屏障语义
- 它不会改变内存内容, 但是会有内存序的副作用

3. 与 LLVM 的集成

如何实现 `__atomic_load_n` / `__atomic_load` ?

- 如果是 `__ATOMIC_RELAXED` 内存序, 则直接使用普通的赋值

```
val = *ptr
```

- 如果是其他任意内存序, 则在赋值后手动添加 `bpf_full_barrier()`

```
val = *ptr;
```

```
bpf_full_barrier()
```


3. 与 LLVM 的集成

如何实现 `__atomic_store_n / __atomic_store` ?

- 如果是 `__ATOMIC_RELAXED` 内存序, 则直接使用普通的赋值

```
*ptr = val
```

- 如果是其他任意内存序, 则在赋值前手动添加 `bpf_full_barrier()`

```
bpf_full_barrier()
```

```
*ptr = val;
```

3. 与 LLVM 的集成

如何实现 `__atomic_test_and_set`?

- 我们可以通过 `__atomic_exchange()` 模式来模拟
`bool old = __atomic_exchange_n(ptr, 1, memorder);`
- 具备 full memory ordering, 与原生 `__atomic_test_and_set` 语义完全一致

3. 与 LLVM 的集成

如何实现 `__atomic_clear` ?

- `__atomic_clear()` 的实质就是一个带顺序语义的 store 操作,即:
`__atomic_store_n(ptr, 0, order)`
- 所以我们直接复用 `__atomic_store_n` 即可

3. 与 LLVM 的集成

如何实现 `__atomic_thread_fence`?

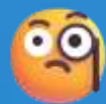
- 如果内存序参数是 `__ATOMIC_RELAXED` 则替换为空操作
- 如果是其他内存序参数那么用 `bpf_full_barrier()` 替代

4.未来方向

- **在指令层面增加 ACQ/REL 指令**
 - 当前的 BPF atomic 指令（如 BPF_FETCH | BPF_ADD）自动保证 full barrier
 - 但在实际场景中并不一定需要全屏障这么强的保证，应用应该能按需选择 memory order
- **Verifier & Analysis Layer — 更好的决策后端**
 - eBPF 的特征之一就是安全，即使是在并发领域！
 - 建立 BPF 指令到 litmus test 转化器，支持 herd7 形式化验证，使得 eBPF 程序更加安全
- **Clang/LLVM 持续支持 API**

4.未来方向

- **在指令层面增加 ACQ/REL 指令**
 - 当前的 BPF atomic 指令（如 BPF_FETCH | BPF_ADD）自动保证 full barrier
 - 但在实际场景中并不一定需要全屏障这么强的保证，应用应该能按需选择 memory order
- **Verifier & Analysis Layer — 更好的决策后端**
 - eBPF 的特征之一就是安全，即使是在并发领域！
 - 建立 BPF 指令到 litmus test 转化器，支持 herd7 形式化验证，使得 eBPF 程序更加安全
- **Clang/LLVM 持续支持 API**



密切关注

Thank you