BPF Memory Model

Presented by Alexei Starovoitov on behalf of Paul McKenney

Overview

- psABI and Memory Model
- BPF Memory Model Context
- BPF Instructions
- JITs Must Respect BPF Memory Model
- Validation: GCC Atomic Built-Ins
- Future Work

processor specific ABI (psABI)

All psABIs define:

- Function calling convention
- Register usage
- Stack usage and unwinding
- Type convention. ex: sizeof(void*)
- ELF object file format
- Relocations and linking
- Libraries
- Code model and address space <- this is not a Memory Model !

Goals of psABIs

x86, RISC-V, and all others psABIs:

- A manual for the compilers: generate compatible binary code

Goals of psABIs

x86, RISC-V, and all others psABIs:

- A manual for the compilers: generate compatible binary code

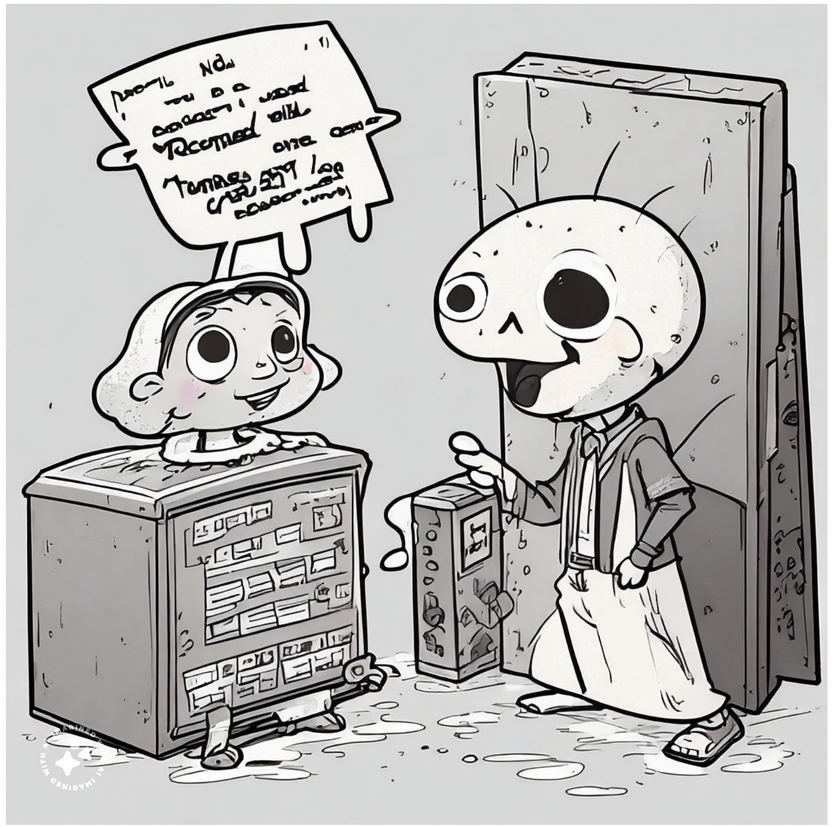
BPF psABI:

- A manual for the compilers: generate compatible binary code
- A manual for JITs: how to map BPF ISA to native ISA

BPF psABI

- It exists !
- JITs use it to translate BPF ISA to native ISA
- GCC and LLVM use it to compile C into BPF assembly
- Largely undocumented
- LLVM/GCC source code is a source of truth
- JITs source code is a source of truth

BPF Memory-Model Context

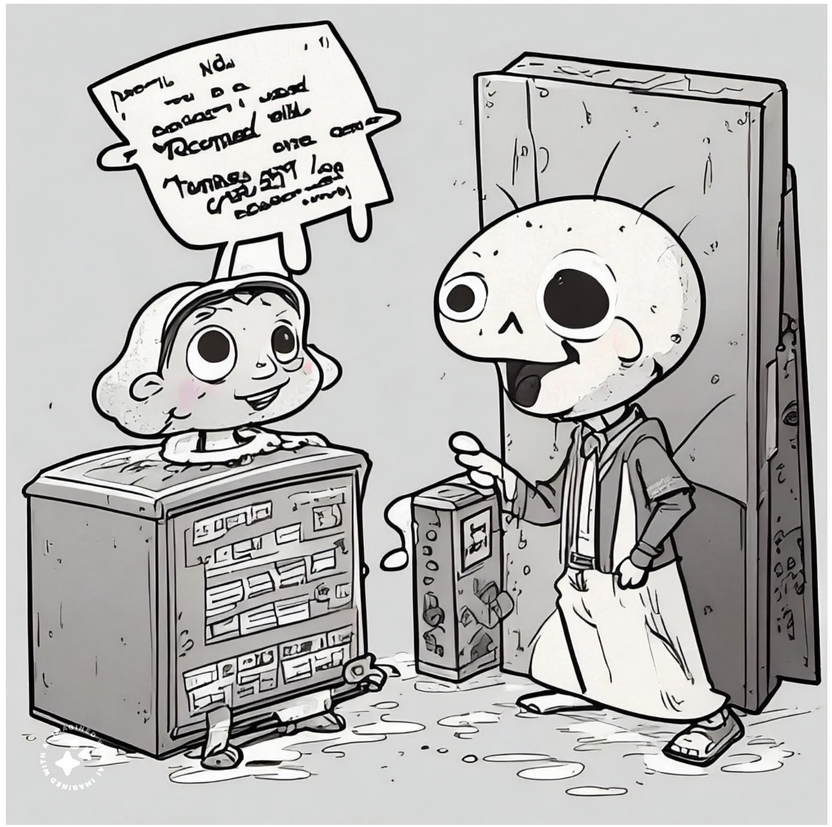


I want to concurrently access data:

1. Between BPF programs
2. Between BPF program and user space
3. Between BPF program and the kernel

How would I do that?

BPF Memory-Model Context



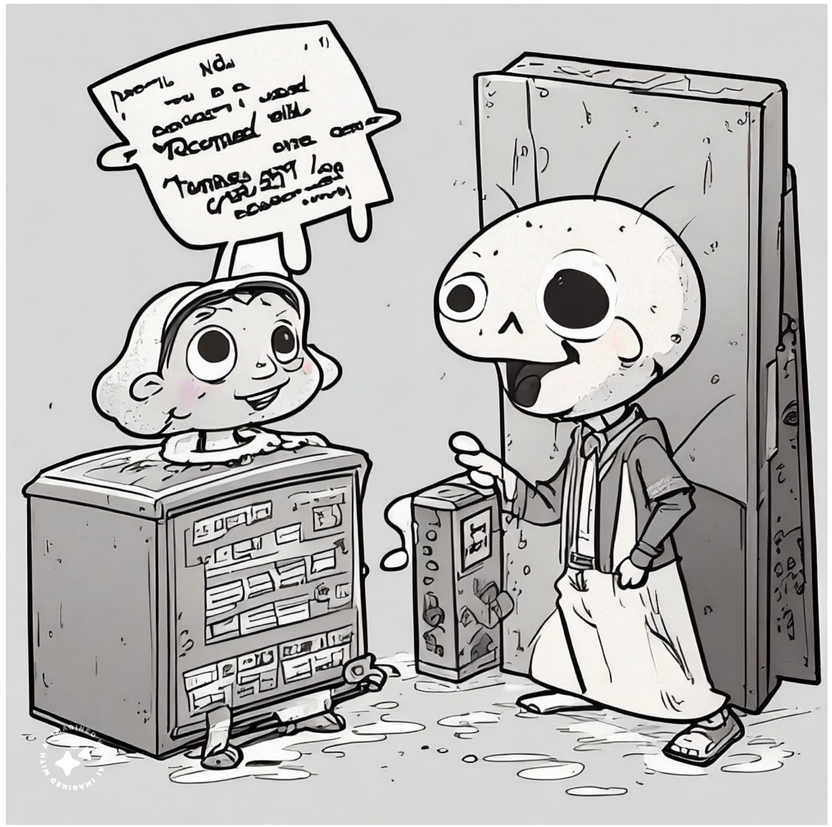
I want to concurrently access data:

1. Between BPF programs
2. Between BPF program and user space
3. Between BPF program and the kernel

How would I do that?

Just use Linux-kernel memory model (LKMM)

BPF Memory-Model Context



I want to concurrently access data:

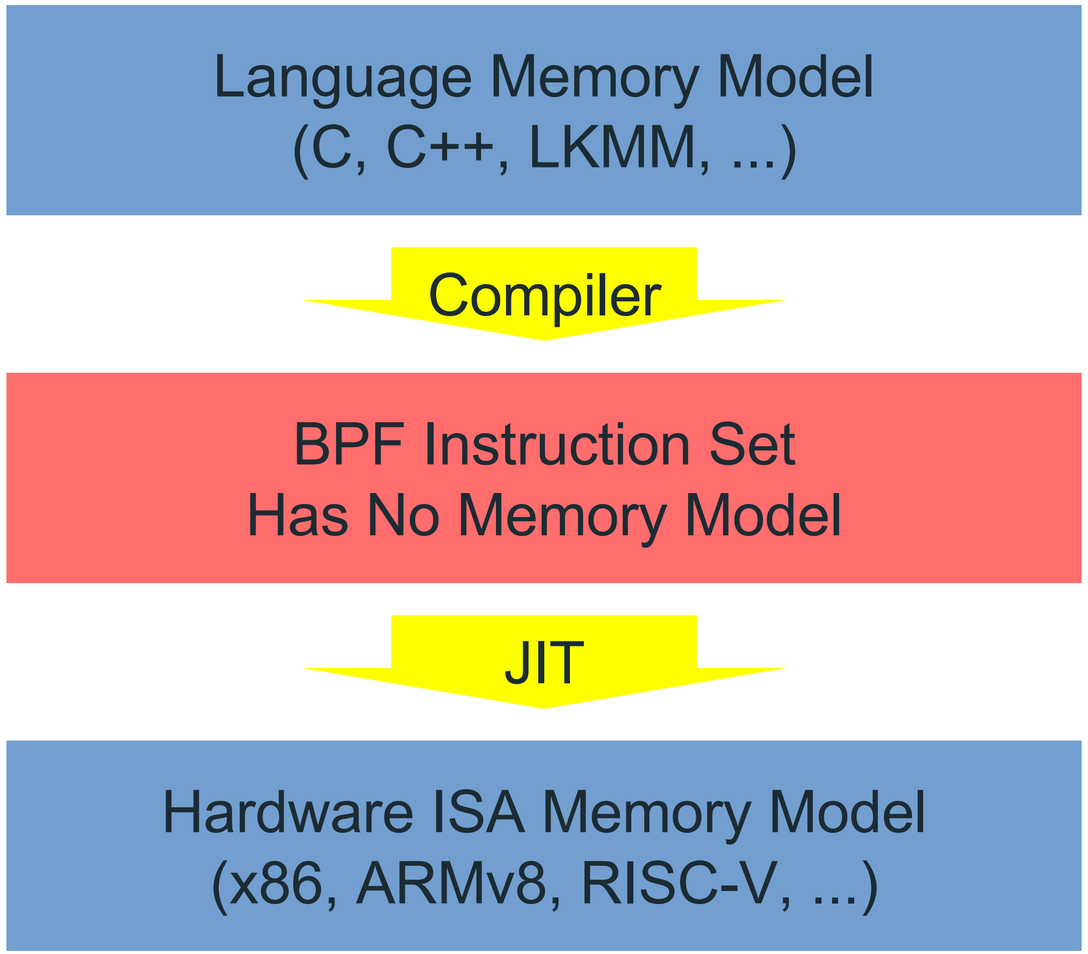
1. Between BPF programs
2. Between BPF program and user space
3. Between BPF program and the kernel

How would I do that?

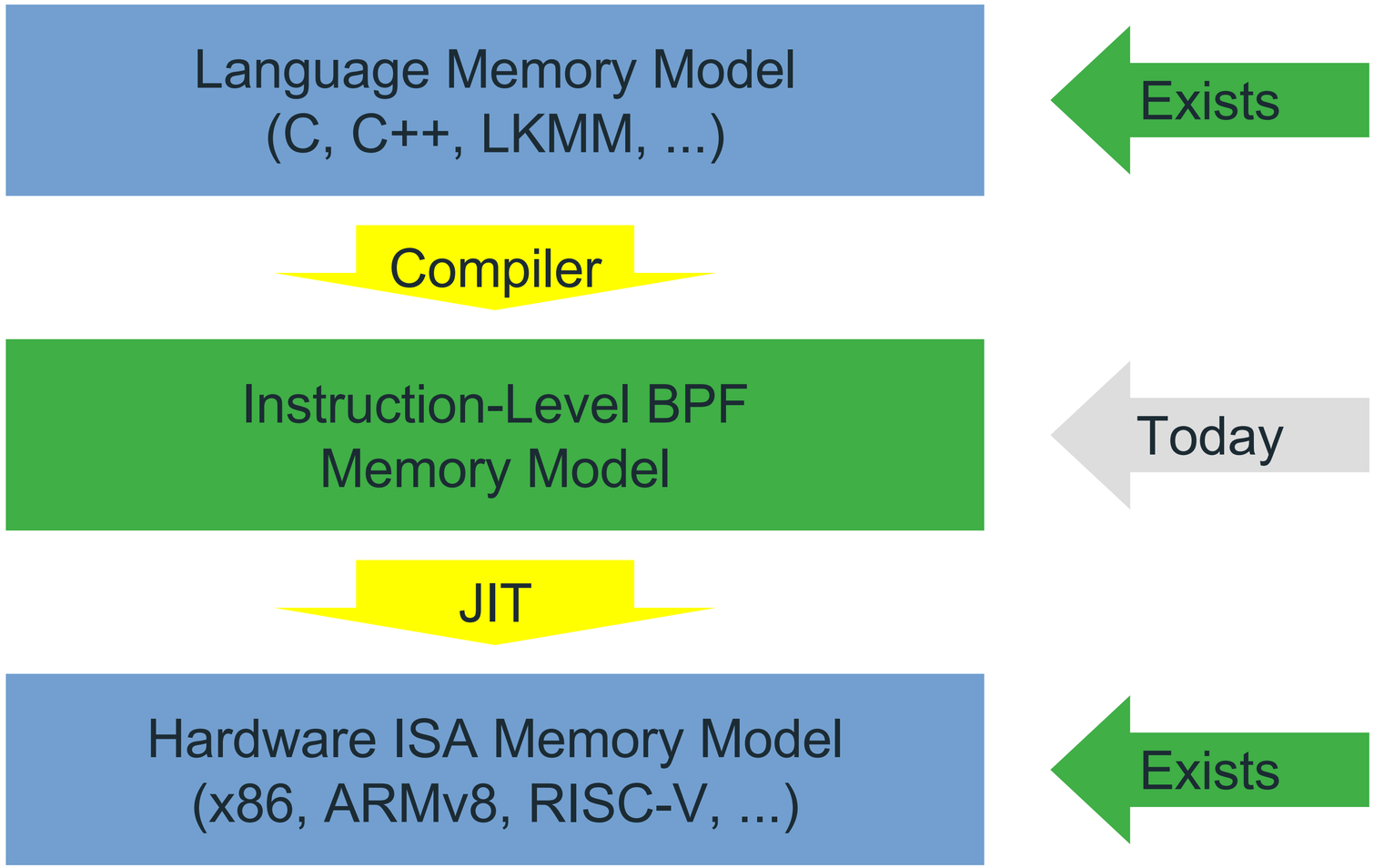
Just use Linux-kernel memory model (LKMM)

Other language MMs are a strict subset of LKMM

Flavors of Memory Models



Flavors of Memory Models



Aside on Linux-Kernel Memory Model

- The Linux kernel uses assembly, C, and Rust
- LKMM relies not just on the language memory model, but also on strict coding conventions:
  - memory-barriers.txt “CONTROL DEPENDENCIES”
  - rcu_dereference.rst
- Language MMs do not handle dependencies
  - And hence are plagued by OOTA issues
  - Therefore, a hardware-level model for BPF instruction set

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1780r0.html
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1916r0.pdf

Example OOTA, x == y == 0 initially

```
r1 = x.load(relaxed);
y.store(r1, relaxed);
```

```
r1 = y.load(relaxed);
x.store(r1, relaxed);
```

Example OOTA, x == y == 0 initially

```
r1 = x.load(relaxed);          r1 = y.load(relaxed);
y.store(r1, relaxed);          x.store(r1, relaxed);
```

According to the mathematical core of the C and C++ memory models, this code can result in x == y == 42!!!

Example OOTA, x == y == 0 initially

```
r1 = x.load(relaxed);          r1 = y.load(relaxed);
y.store(r1, relaxed);          x.store(r1, relaxed);
```

According to the mathematical core of the C and C++ memory models, this code can result in x == y == 42!!!

But only in theory, not in practice!

Defining BPF Memory Model

- It exists !
- Largely undocumented
- LLVM source code is a source of truth
- JITs and interpreter source code is a source of truth

BPF Instructions

- BPF Atomic Instructions
- BPF Conditional Jump Instructions
- BPF Load instructions

BPF Atomic Instructions

- BPF_XCHG, BPF_CMPXCHG
- BPF_ADD, BPF_OR, BPF_AND, BPF_XOR
- BPF_FETCH with one of the above

BPF Atomic Instructions 1/3

- BPF_XCHG and BPF_CMPXCHG instructions are fully ordered
- All CPUs and tasks agree that all instructions preceding or following a given BPF_XCHG or BPF_CMPXCHG instruction are ordered before or after, respectively, that same instruction
  - Consistent with Linux-kernel atomic_xchg() and atomic_cmpxchg(), respectively
  - Alternatively, consistent with the following:
    - smp_mb(); atomic_cmpxchg_relaxed(); smp_mb();

BPF Atomic Instructions 2/3

- BPF_ADD, BPF_OR, BPF_AND, BPF_XOR instructions are unordered
- CPUs and JITs can reorder these instructions freely
  - Consistent with Linux-kernel atomic_add(), atomic_or(), atomic_and(), and atomic_xor() APIs

BPF Atomic Instructions 3/3

- When accompanied by BPF_FETCH, BPF_ADD, BPF_OR, BPF_AND, BPF_XOR instructions are fully ordered
- All CPUs and tasks agree that all instructions preceding or following a given instruction adorned with BPF_FETCH are ordered before or after, respectively, that same instruction
  - Consistent with Linux-kernel atomic_fetch_add(), atomic_fetch_or(), atomic_fetch_and(), and atomic_fetch_xor() APIs

BPF Conditional Jump Instructions

- Modifiers to BPF_JMP32 and BPF_JMP instructions:
  - BPF_JEQ, BPF_JGT, BPF_JGE, BPF_JSET, BPF_JNE, BPF_JSGT, BPF_JSGE, BPF_JLT, BPF_JLE, BPF_JSLT, and BPF_JSLE
- Unconditional jump instructions (BPF_JA, BPF_CALL, BPF_EXIT) provide no memory-ordering semantics

BPF Conditional Jump Instructions

- These modifiers to BPF_JMP32 and BPF_JMP instructions provide weak ordering:
  - BPF_JEQ, BPF_JGT, BPF_JGE, BPF_JSET, BPF_JNE, BPF_JSGT, BPF_JSGE, BPF_JLT, BPF_JLE, BPF_JSLT, and BPF_JSLE
- Too-smart JITs might need to be careful

BPF Conditional Jump Instructions

- This weak ordering applies when:
  - Either the src or dst registers depend on a prior load instruction (BPF_LD or BPF_LDX), ***and***
  - There is a store instruction (BPF_ST or BPF_STX) *before control flow converges*, ***and***
  - The restrictions outlined in the “CONTROL DEPENDENCIES” section of Documentation/memory-barriers.txt are faithfully followed
    - Compilers do not understand control dependencies, and happily break them.
    - Optimizations involving conditional-move instructions requires the “before control flow converges” restriction

Conditional Jump Example

```
r1 = READ_ONCE(x);
if (r1) {
        WRITE_ONCE(y, 42);
}
```


```
        r1 = x ll
        r1 = *(u64 *)(r1 + 0)
        if r1 == 0 goto LBB0_2
        r1 = y ll
        *(u64 *)(r1 + 0) = r2
LBB0_2:
```

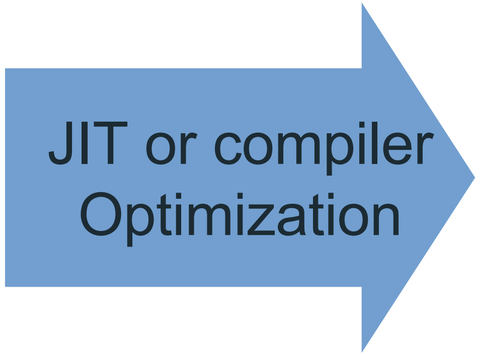
https://godbolt.org/z/hh7n9eqrv

Control-Dependency Breakage

```
r0 = READ_ONCE(*x);
if (r0) {
    WRITE_ONCE(*y, 1);
} else {
    WRITE_ONCE(*y, 1);
}
```

Control-Dependency Breakage

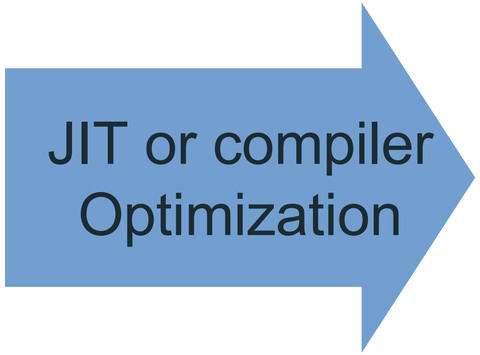
```
r0 = READ_ONCE(*x);
if (r0) {
    WRITE_ONCE(*y, 1);
} else {
    WRITE_ONCE(*y, 1);
}
```




```
r0 = READ_ONCE(*x);
WRITE_ONCE(*y, 1);
```

Control-Dependency Breakage

```
r0 = READ_ONCE(*x);
if (r0) {
    WRITE_ONCE(*y, 1);
} else {
    WRITE_ONCE(*y, 1);
}
```




```
r0 = READ_ONCE(*x);
WRITE_ONCE(*y, 1);
```

Broken Dependency!!!

BPF Conditional Jump Instructions

- Different hardware architectures order control dependencies in different ways:
  - Strongly ordered (x86, s390, …):
    - Prior load instructions are ordered before later store instructions, courtesy of TSO
  - Weakly ordered (ARMv8, PowerPC, …):
    - Control dependencies are tracked by hardware

BPF Conditional Jump Instructions

- What do you mean by “weak”???
  - CPU 0’s control dependency is visible to CPU 1, and separately to CPU2
  - But CPU 0’s control dependency is not necessarily visible to code spanning CPU 1 and 2

Example of Weakness in Play

CPU 0

```
WRITE_ONCE(*x, 1);
```

CPU 1

```
r0 = READ_ONCE(*x);
if (r0) {
   // Control dependency
   WRITE_ONCE(*y, 1);
}
```

CPU 2

```
r0 = smp_load_acquire(y);
r1 = READ_ONCE(*x);
```

Both r0 instances and r1 can all be zero!!!

Example For Converging Control Flow

```
r1 = READ_ONCE(x);
if (r1)
    WRITE_ONCE(y, 1);
else
    WRITE_ONCE(y, 2);
WRITE_ONCE(z, 1); // Converged here
```

JIT uses cmov

```
mov r1,(x)
mov r2,$2
cmov r1,r2,$1
mov (y),r2
mov (z),$1
```

Control dependency constrains ordering

No ordering constraint!!!

Which is why control dependencies extend only to control-flow convergence!!!

BPF Load Instructions

- BPF_LD and BPF_LDX instructions
  - If the value returned by a given load instruction is used to compute the address of a later load or store instruction, address-dependency ordering is guaranteed
  - If the value returned by a given load instruction is used to compute the value stored by a given store instruction, data-dependency ordering is guaranteed
  - These are used by RCU readers, which must faithfully follow the restrictions outlined in Documentation/RCU/rcu_dereference.rst
    - Compilers do not understand address or data dependencies, and happily break them.
    - Address and data dependencies are weak, similar to control dependencies

BPF Load Instructions

- Different hardware architectures order address and data dependencies in different ways:
  - Strongly ordered (x86, s390, ...):
    - Prior load instructions are ordered before later load and store instructions, courtesy of TSO
  - Weakly ordered (ARMv8, PowerPC, ...):
    - Address and data dependencies are tracked by hardware

Example of Weakness in Play

CPU 0

```
WRITE_ONCE(*x, 1);
```

CPU 1

```
r0 = READ_ONCE(*x);
// Data dependency
WRITE_ONCE(*y, r0);
```

CPU 2

```
r0 = smp_load_acquire(y);
r1 = READ_ONCE(*x);
```

Both r0 instances and r1 can all be zero!!!

JITs must respect BPF Memory Model

- Viable strategies:
  - Preserve address, control, and data dependencies
    - Just generate instructions that match the BPF assembly code most closely
    - Put atomic_signal_fence(memory_order_seq_cst) everywhere
    - Trace and explicitly preserve dependencies
  - Order prior loads before later stores:
    - JIT every BPF_LD and BPF_LDX into a target-machine load-acquire instruction sequence
    - Place at least one target-machine load-to-store memory-barrier instruction between each BPF load/store instruction pair
      - atomic_signal_fence(memory_order_acquire) works on x86
  - Rely on source-level code having followed Linux-kernel coding standards

Discovering BPF Memory Model via GCC Atomic Built-Ins

Note that GCC defined the built-ins, but this section uses only Clang/LLVM

GCC Atomic Memory Orders

- __ATOMIC_RELAXED: Relaxed ordering
- __ATOMIC_ACQUIRE: Acquire ordering
- __ATOMIC_CONSUME: Treated as acquire
- __ATOMIC_RELEASE: Release ordering
- __ATOMIC_ACQ_REL: Acquire/release ordering
- __ATOMIC_SEQ_CST: Sequential consistency

No BPF C/C++ Weak Orderings

- __ATOMIC_RELAXED: Relaxed ordering
- __ATOMIC_ACQUIRE, __ATOMIC_CONSUME, __ATOMIC_RELEASE, __ATOMIC_ACQ_REL, __ATOMIC_SEQ_CST: Full ordering
- Revisit when BPF does acquire and release

GCC Full Memory Barriers

- __atomic_thread_fence(__ATOMIC_SEQ_CST)
- BPF has none, but it can emulate them:
  - “BPF_ATOMIC | BPF_DW | BPF_STX” with an imm field of “BPF_ADD | BPF_FETCH” and a src register value of zero
  - Or: “lock *(u32 *)(r2 + 0) += r1”
  - Call it bpf_mb() for short

GCC Atomic Loads

- __atomic_load_n() & __atomic_load()
- Relaxed ordering:
  - BPF_LD or BPF_LDX
- Non-relaxed ordering:
  - BPF_LD or BPF_LDX followed by bpf_mb()

Clang/LLVM does not yet support this

GCC Atomic Stores

- __atomic_store_n() & __atomic_store()
- Relaxed ordering:
  - BPF_ST or BPF_STX
- Non-relaxed ordering:
  - bpf_mb() followed by BPF_ST or BPF_STX

Clang/LLVM does not yet support this

GCC Atomic Exchange

- __atomic_exchange_n() & __atomic_exchange()
- No matter what ordering:
  - “BPF_ATOMIC | BPF_DW | BPF_STX'' with an immediate field of “BPF_XCHG | BPF_FETCH”, which supplies full ordering

GCC Atomic Compare and Exchange

- __atomic_compare_exchange_n() & __atomic_compare_exchange()
- No matter what ordering:
  - “BPF_ATOMIC | BPF_DW | BPF_STX'' with an immediate field of “BPF_CMPXCHG | BPF_FETCH”, which supplies full ordering

GCC Atomic Fetch-Op

- __atomic_fetch_add(), __atomic_fetch_sub(), __atomic_fetch_and(), __atomic_fetch_xor()
  - “BPF_ATOMIC | BPF_DW | BPF_STX'' with an immediate field of “BPF_XXX | BPF_FETCH”, which supplies full ordering, needed or not
  - Where “XXX” is ADD, SUB, AND, and XOR, respectively
- __atomic_fetch_or(), __atomic_fetch_nand()
  - Loop containing “BPF_ATOMIC | BPF_DW | BPF_STX'' with an immediate field of “BPF_CMPXCHG | BPF_FETCH”, which supplies full ordering, needed or not
  - Use BPF_OR or a combination of BPF_AND with best bit-complement code, respectively

Clang/LLVM does not yet support __atomic_fetch_nand()

GCC Atomic Op-Fetch

- __atomic_add_fetch(), __atomic_sub_fetch(), __atomic_and_fetch(), __atomic_xor_fetch(), __atomic_or_fetch(), __atomic_nand_fetch()
  - Implement in the same way as for atomic_fetch_xxx()
  - Except that it is necessary to fix up return value to provide the after-operation value
  - Full ordering is supplied whether it is needed or not

Clang/LLVM does not yet support __atomic_nand_fetch()

GCC Miscellaneous Atomics

- __atomic_test_and_set()
  - Implement the same as __atomic_exchange()
  - Except casting the return value to boolean if needed
- __atomic_clear()
  - Implement as an __atomic_store() of zero

Clang/LLVM does not yet support these

GCC Fences

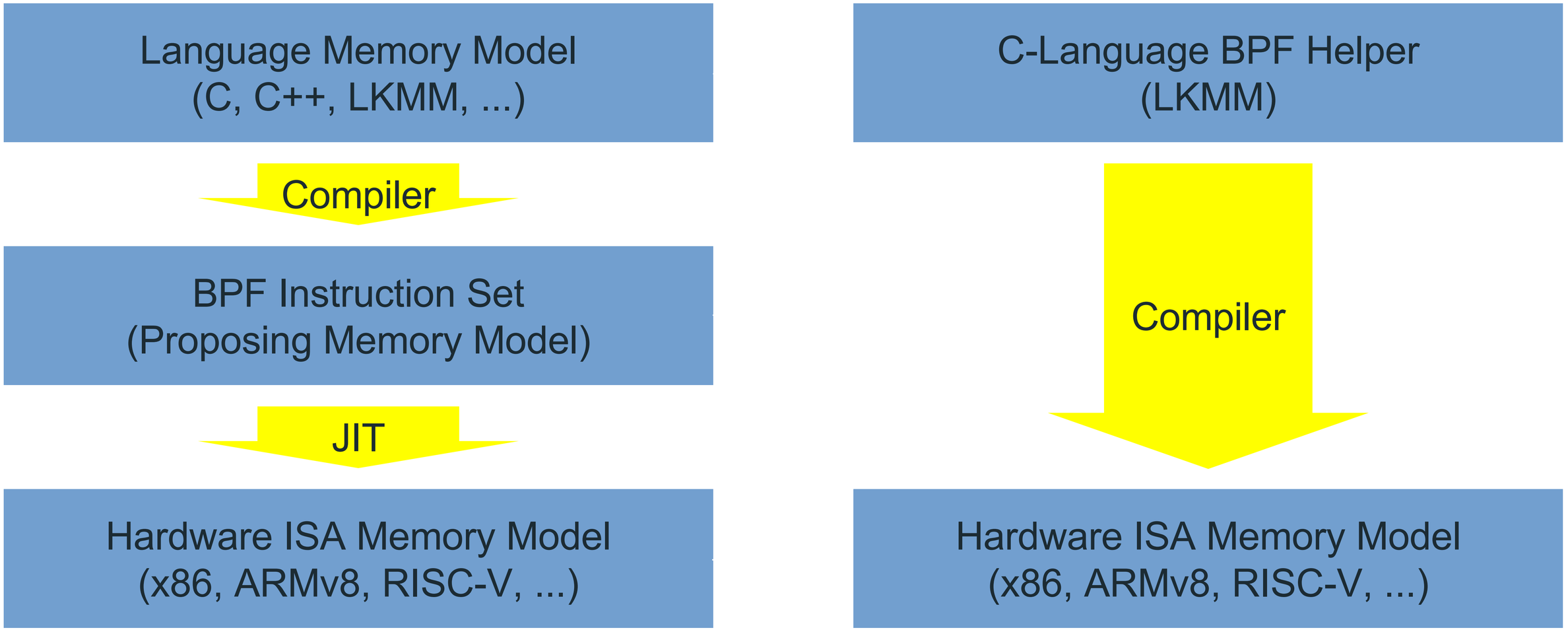
- __atomic_thread_fence()
  - Implement as bpf_mb()
- __atomic_signal_fence()
  - Implement as the Linux-kernel barrier() macro
  - Unless relaxed, in which case this is a no-op

Clang/LLVM does not yet support these

Complication: BPF Helper Ordering?

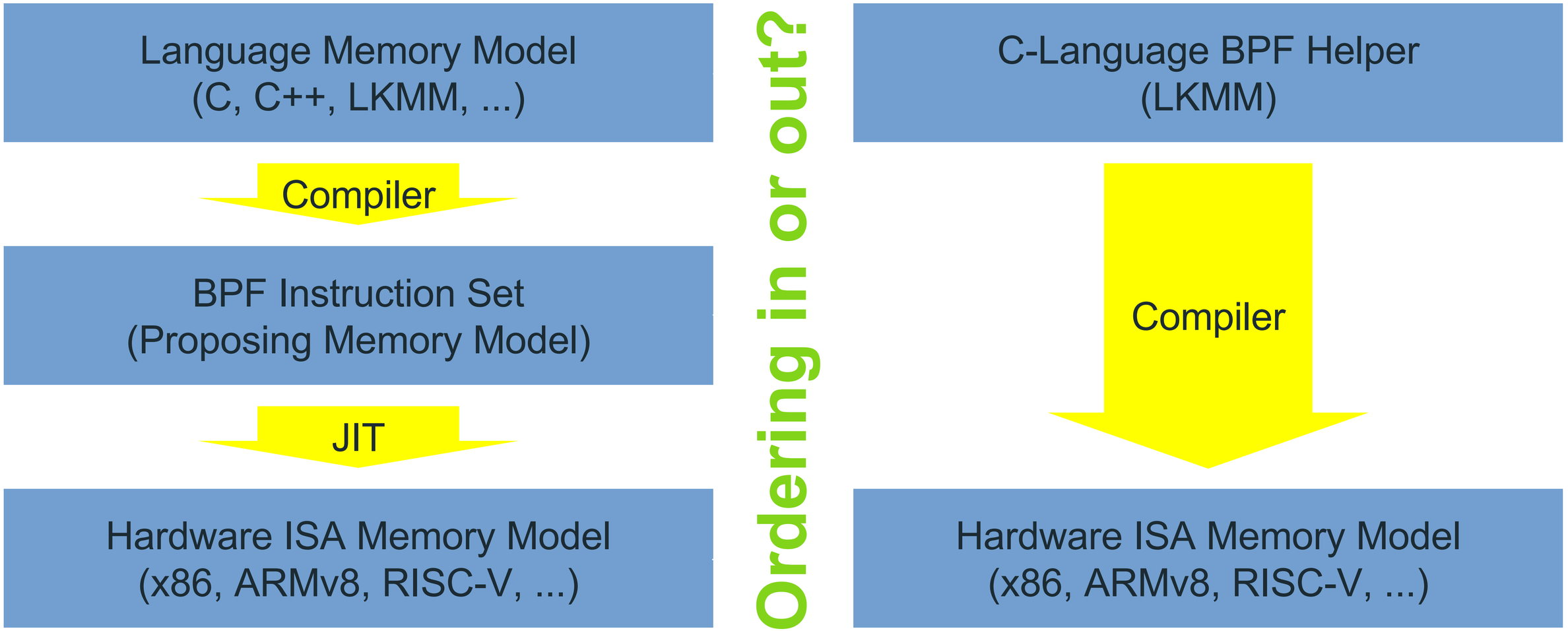
https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

Complication: BPF Helper Ordering?



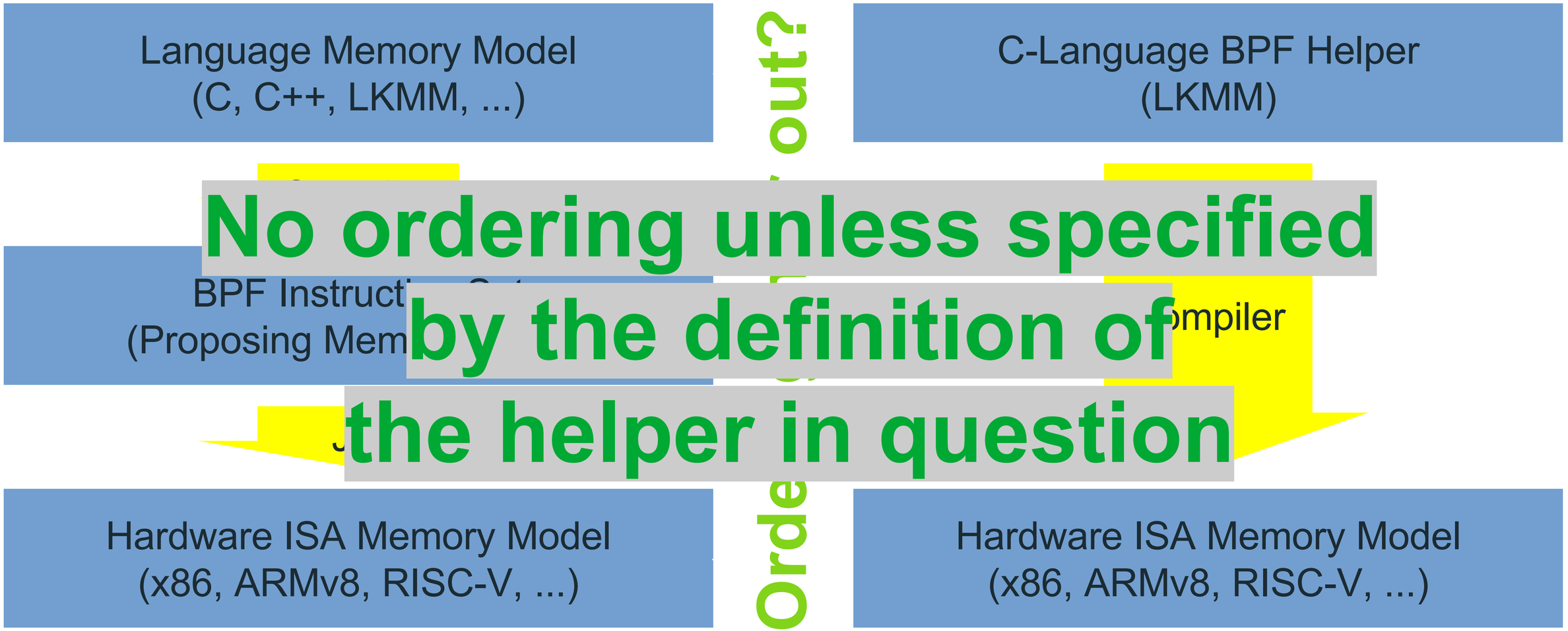
https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

Complication: BPF Helper Ordering?



https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

Complication: BPF Helper Ordering?



https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

BPF ISA extensions

- Possible additions:
  - BPF_LDX_ACQ - load acquire
  - BPF_STX_REL - store release store
  - Full barrier
    - Possibly one variant with I/O semantics and another variant having only normal-memory semantics
    - Normal-memory semantics (smp_mb()) is more urgent

For More Information

- “Instruction-Level BPF Memory Model”
  - https://docs.google.com/document/d/1TaSEfWfLnRUi5KqkavUQyL2tThJXYWHS15qcbxIsFb0/edit?usp=sharing
- “IETF eBPF Instruction Set Specification, v1.0”
  - https://www.ietf.org/archive/id/draft-thaler-bpf-isa-00.html
- “Towards a BPF Memory Model” (2021 BPF & Networking Summit at Linux Plumbers Conference)
  - https://lpc.events/event/11/contributions/941/
- “”GCC Atomic Compiler Built-Ins”
  - https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html
- Linux kernel source tree: tools/memory-model, Documentation/memory-barriers.txt “CONTROL DEPENDENCIES” section, Documentation/RCU/rcu_dereference.rst
- “Is Parallel Programming Hard, And, If So, What Can You Do About It?”
  - Chapter 15 (“Advanced Synchronization: Memory Ordering”)
  - Appendic C (“Why Memory Barriers?”)
    - https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html