

blitmus: litmus-testing the BPF memory model on real hardware

Puranjay Mohan

github.com/puranjaymohan/blitmus



```
void cpu0(int *x, int *y)
{
    WRITE_ONCE(*x, 1);
    r0 = READ_ONCE(*y);
}
```

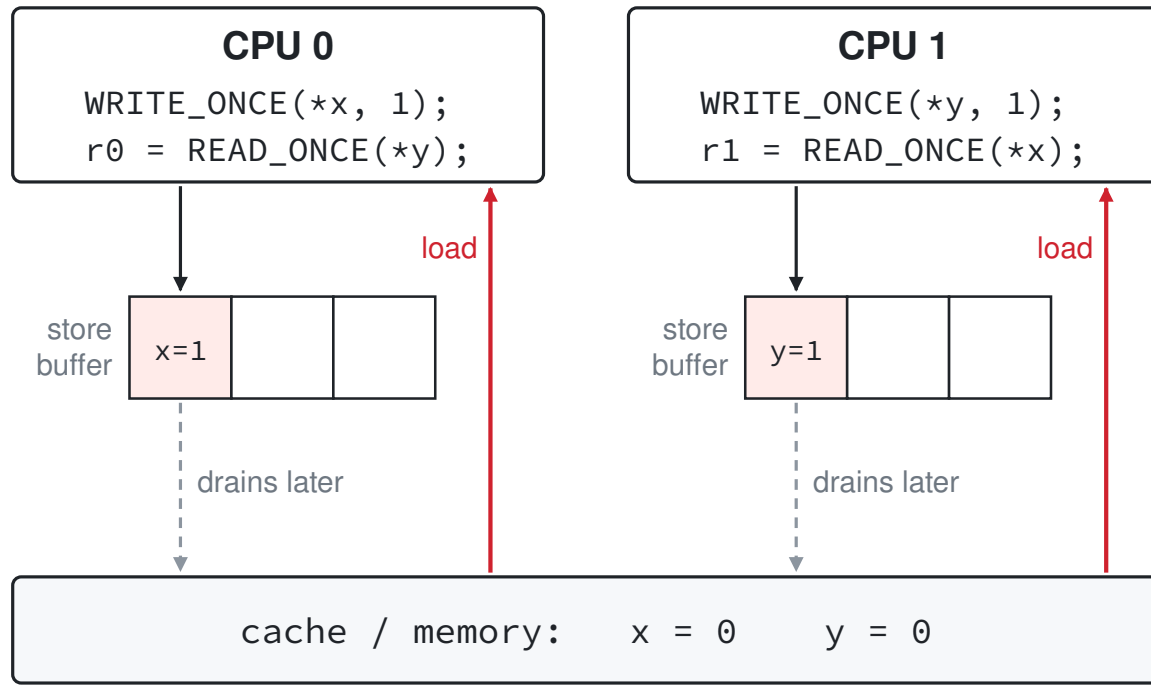
```
void cpu1(int *x, int *y)
{
    WRITE_ONCE(*y, 1);
    r1 = READ_ONCE(*x);
}
```

Can r0 and r1 both be 0?

*x and *y start at 0. Each CPU raises its own flag, then checks the other's.
If both read 0, both think they're alone. Hands up for yes.



Yes. Almost half the time.



each load overtakes its own CPU's store: $r0 = r1 = 0$

arm64

46.3%

18,993,260 of 41M runs

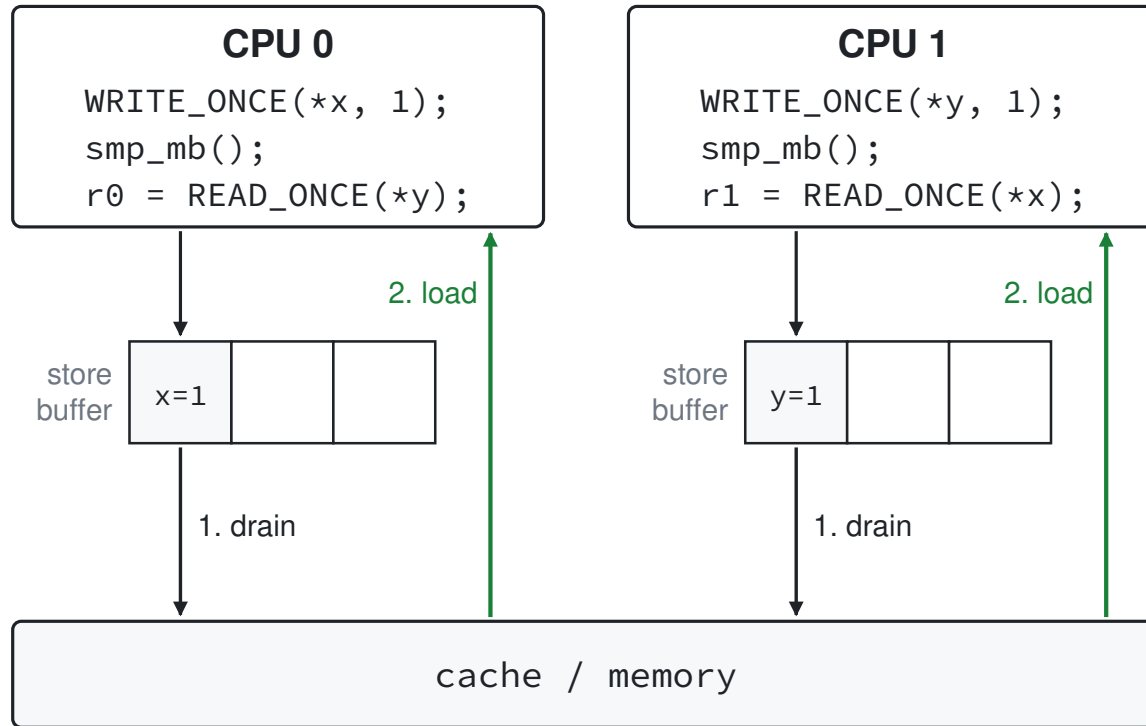
x86

4.1%

1,685,991 of 41M runs

One run: both CPUs execute their two lines once, at the same time. blitmus runs this as BPF programs, one per CPU, 41 million times per machine.

Unless you ask for ordering



each load waits for its own store: at least one CPU reads 1

`smp_mb()` is a promise: the store is visible before the load. In kernel C it's arch asm that the compiler can't touch. In BPF the promise passes through more hands.

arm64, with `smp_mb()`

0

of 41M runs

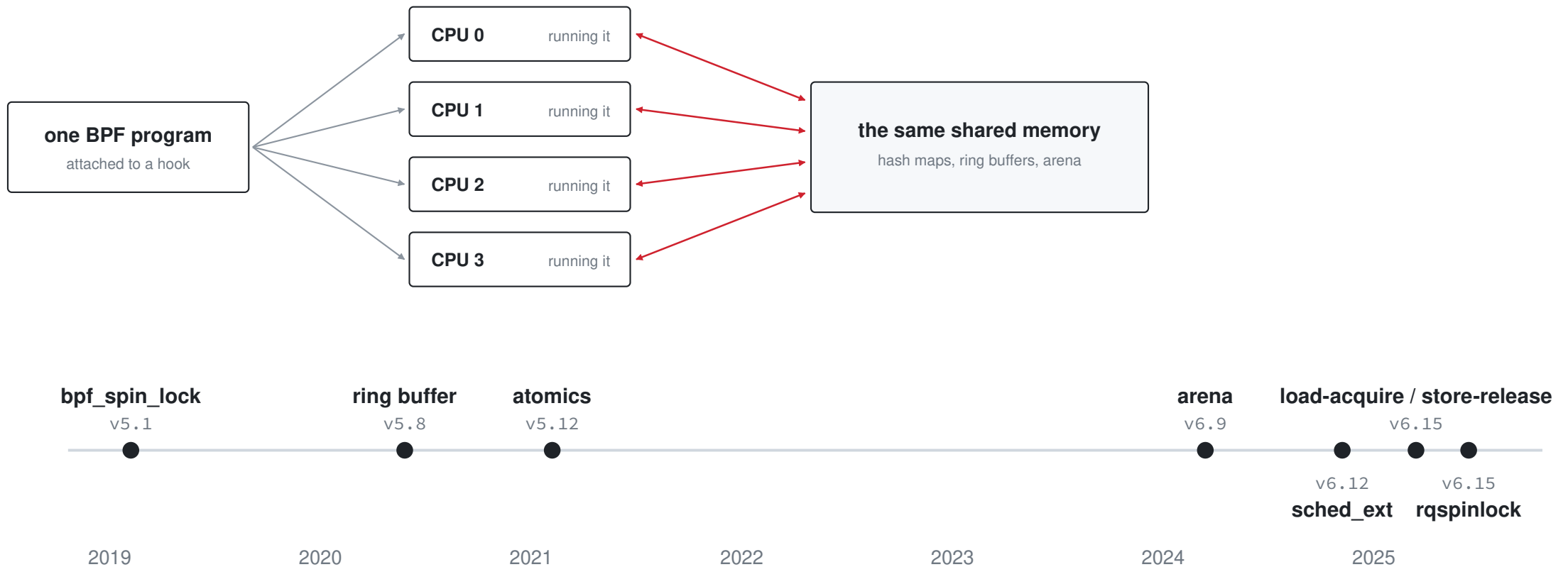
x86, with `smp_mb()`

0

of 41M runs

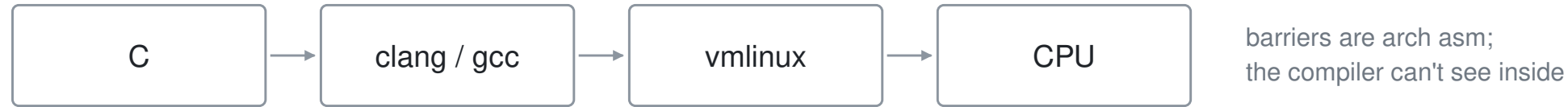
BPF is concurrent code now

In tree: `scx_flatcg` coordinates its callbacks across CPUs with compare-and-swap and atomic counters; `scx_sdt` keeps per-task data in arena pages.

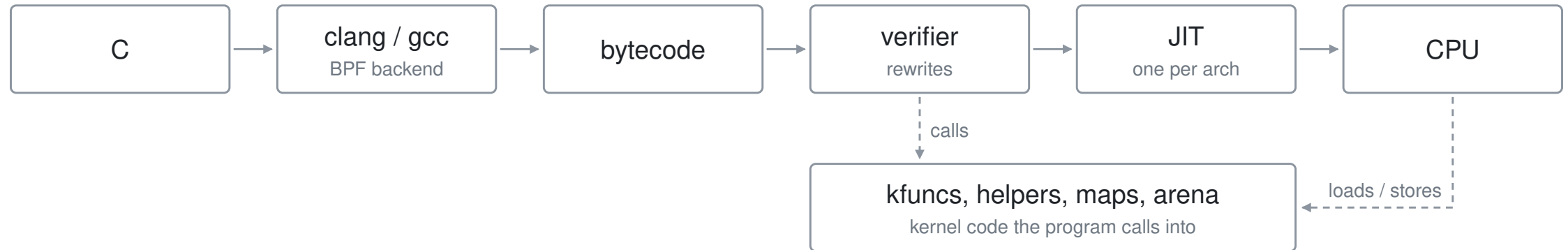


Who keeps the promise?

kernel C



BPF



Every link in the BPF chain has to preserve the ordering the program asked for. If one drops it, the program can't tell, and neither can a functional test.

Three links that broke

BUG 1 · COMPILER

An `smp_mb()` that does nothing

where clang / gcc BPF backend
breaks arm64, riscv, powerpc
fixed LLVM 20, GCC 15

1 run in 7

stress test, arm64

BUG 2 · JIT

Fully ordered atomics that weren't

where powerpc and riscv JITS
breaks powerpc, riscv
fixed v6.10

1 run in 6

stress test, powerpc

BUG 3 · KFUNC

One address, two pages

where arena page alloc / free
breaks any arch; seen on x86, arm64
fixed series on the list

25 in 5 million

stress test, arm64

Every functional test passed with all three, and none of them is a bug in a BPF program.



Bug 1: an `smp_mb()` that does nothing



BPF has no fence instruction, so BPF code gets `smp_mb()` from an atomic that returns a value, and ignores the value:

BPF C	<code>atomic_fetch_add(&z, 1);</code>	<code>/* result unused: it's only a barrier */</code>
clang 19	<code>lock *(u64 *)(r1 + 0) += r3</code>	<code>/* FETCH dropped */</code>
arm64 JIT	<code>stadd</code>	<code>/* no ordering */</code>
riscv JIT	<code>amoadd, aq = rl = 0</code>	<code>/* no ordering */</code>
powerpc JIT	<code>ldarx / add / stdcx., no sync</code>	<code>/* no ordering */</code>
x86 JIT	<code>lock add</code>	<code>/* still a full barrier */</code>

The store before the "barrier" and the load after it can pass each other on every JIT but x86. No selftest checked the ordering, so CI passed on every architecture.

Bug 1: why, and the numbers



clang $\leq$ 19 and gcc $<$ 15 dropped `FETCH` when the result was unused. For normal C that's fine: nobody uses an atomic as a barrier there, they use a fence. BPF has no fence, so BPF code does, and in BPF the add without `FETCH` is relaxed. Changed in LLVM 20 and GCC 15.

```
WRITE_ONCE(*x, 1);  
atomic_fetch_add(z, 1);  
r0 = READ_ONCE(*y);  
  
WRITE_ONCE(*y, 1);  
atomic_fetch_add(z, 1);  
r1 = READ_ONCE(*x);
```

the question, with an atomic as the barrier; $r0 = r1 = 0$ must never happen

arm64, built with clang 19

14.7%

6,038,875 of 41M runs

X violated

same machine, clang trunk

0

of 41M runs

✓ holds

Bug 2: fully ordered atomics that weren't



A BPF atomic with `FETCH` is supposed to be fully ordered, like the kernel's `atomic_fetch_add()`. On x86 every atomic is `locked`; the arm64 JIT emits `ldaddal`. Two JITs emitted a relaxed RMW:

powerpc, `bpf_jit_comp64.c`

```
EMIT(PPC_RAW_LI(tmp1_reg, off));
+if ((imm & BPF_FETCH) && IS_ENABLED(CONFIG_SMP))
+  EMIT(PPC_RAW_SYNC());
tmp_idx = ctx->idx * 4;
@@ lwarx / op / stwcx. / bne loop @@
if (imm & BPF_FETCH) {
+  if (IS_ENABLED(CONFIG_SMP))
+    EMIT(PPC_RAW_SYNC());
  EMIT(PPC_RAW_MR(ret_reg, _R0));
```

riscv, `bpf_jit_comp64.c`

```
case BPF_ADD | BPF_FETCH:
-  emit(is64 ? rv_amoadd_d(rs, rs, rd, 0, 0) :
-        rv_amoadd_w(rs, rs, rd, 0, 0), ctx);
+  emit(is64 ? rv_amoadd_d(rs, rs, rd, 1, 1) :
+        rv_amoadd_w(rs, rs, rd, 1, 1), ctx);
```

20a759df3bba: set `aq` and `rl` on the AMO; same for the other `FETCH` ops and `XCHG`

b1e7cee96127: `sync` on both sides of the LL/SC loop

Bug 2: why nobody noticed



The kernel's own `atomic_fetch_add()` on powerpc and riscv was always correct; the JITs carry their own copy of that mapping, and theirs was wrong. The arithmetic was right, so every functional test passed. Both were added in 2022 and fixed in v6.10.

```
atomic_fetch_add(x, 1);      atomic_fetch_add(y, 1);  
r0 = READ_ONCE(*y);         r1 = READ_ONCE(*x);
```

the question, with each store an atomic; `r0 = r1 = 0` must never happen

powerpc, JIT without the fix

17.3%

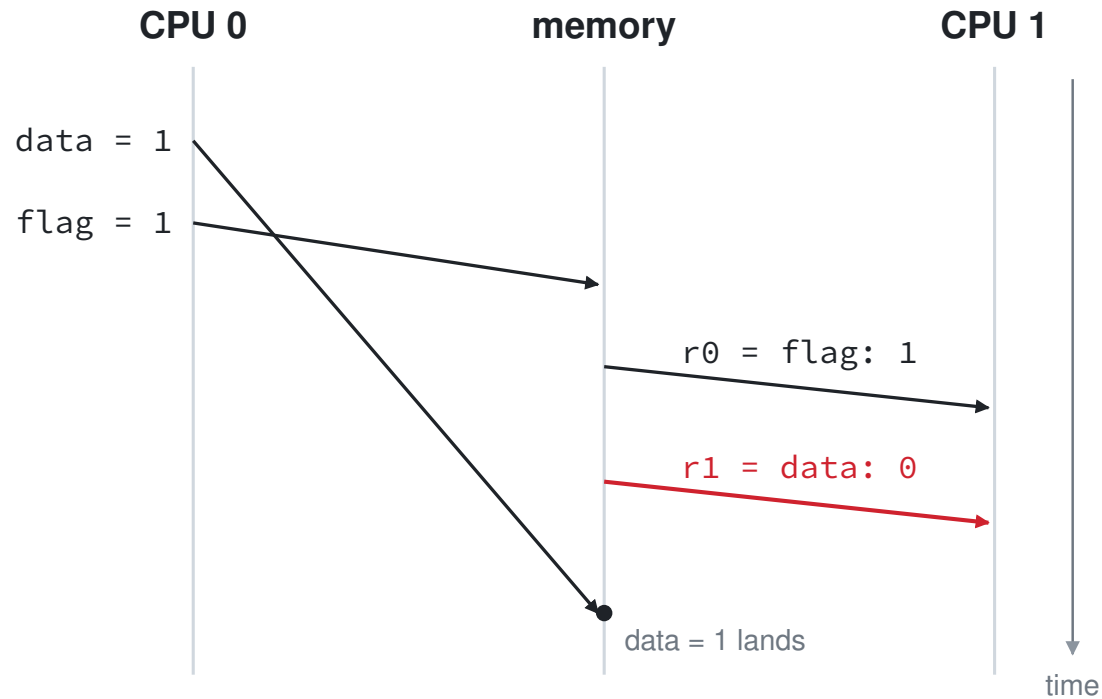
7,085,332 of 41M runs
X violated

powerpc, with b1e7cee96127

0

of 41M runs
✓ holds

Publishing data: write it, then raise a flag



`r0 = 1, r1 = 0`: CPU 1 sees the flag before the data

arm64

7.3%

2,974,897 of 41M runs

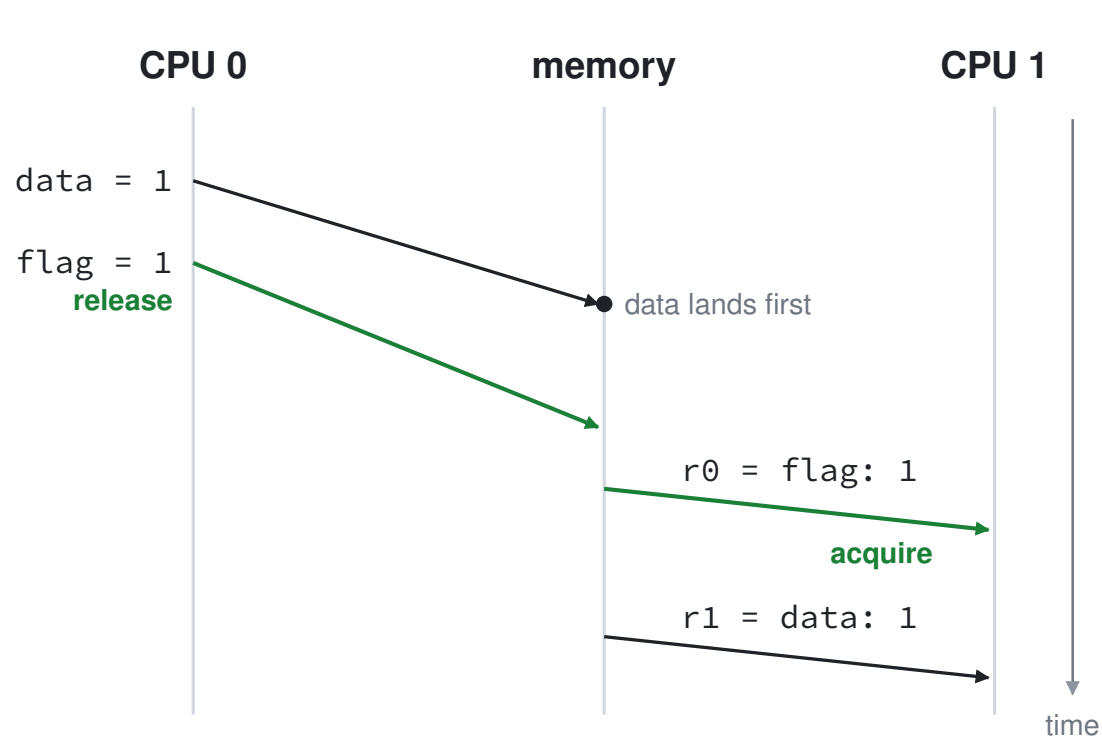
x86

0

of 41M runs: x86 keeps stores and loads in order

The other classic shape: one CPU writes the data and then the flag, the other reads the flag and then the data. On arm64 the reader can see the flag and the old data.

Release the flag, acquire it



the flag can't overtake the data, on either side

arm64, release / acquire

0

of 41M runs

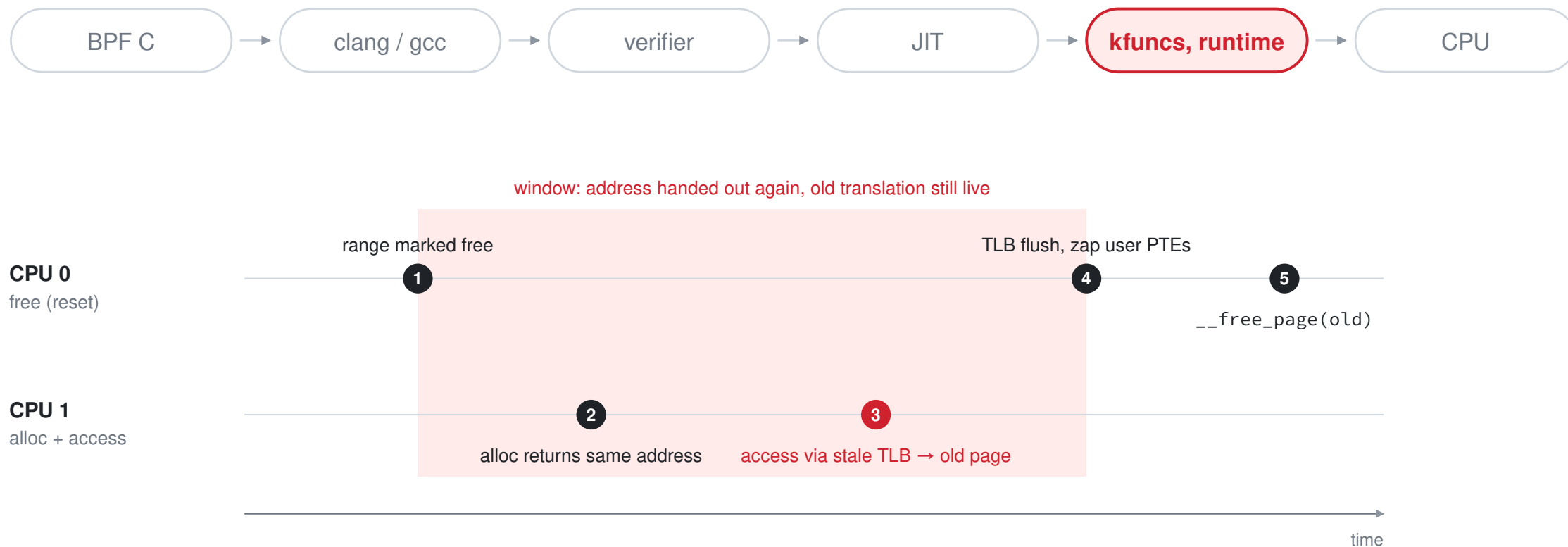
x86, release / acquire

0

of 41M runs

`smp_store_release()` keeps the data ahead of the flag and `smp_load_acquire()` keeps the data read behind it. See the flag, and you see the data. Bug 3 is this exact code, failing anyway.

Bug 3: one address, two pages



The free path handed the address out again before flushing the old translation. The fix (v3 on the list) keeps a freed range unavailable until the TLB flush and the user PTE zap are done.

Bug 3: the numbers



```
/* CPU 0 */
p = bpf_arena_alloc_pages(...);
WRITE_ONCE(*p, 1);
page[i] = p;
smp_store_release(&flag, 1);

/* CPU 1 */
r0 = smp_load_acquire(&flag);
p = page[i];
r1 = p ? READ_ONCE(*p) : 0;

/* between rounds: bpf_arena_free_pages() on every page */
```

the flag is set, so the data must be there; $r0 = 1$, $r1 = 0$ must never happen

The program, the compiler and the JIT were all fine. With full barriers instead of release/acquire it still fails: it isn't an ordering bug, it's two CPUs seeing two pages.

arm64, bpf-next

25

of 5.12M runs (x86: 11)

X violated

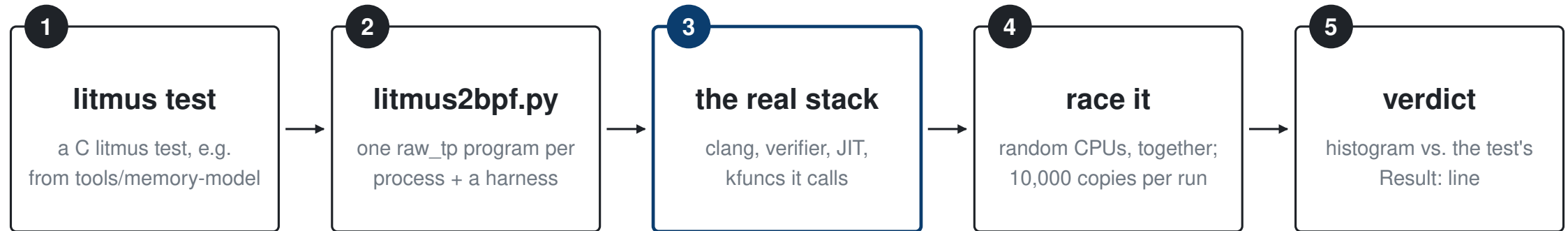
arm64, with the series

0

of 5.12M runs

✓ holds

How we catch them: blitmus



the test says **Never**, and blitmus sees it → **something in the BPF stack broke the ordering**

blitmus turns any litmus test into a BPF race test. Next: one test end to end, with the real commands and their real output. SB+fetchadd, the test behind bug 2.

Step 1: pick a litmus test

```
$ cat litmus_tests/SB+fetchadd.litmus
C SB+fetchadd
(*
 * Result: Never
 * ...
 *)
{}
P0(int *x, int *y)
{
    int r0;
    atomic_fetch_add(x, 1);
    r0 = READ_ONCE(*y);
}
P1(int *x, int *y)
{
    int r0;
    atomic_fetch_add(y, 1);
    r0 = READ_ONCE(*x);
}
exists (0:r0=0 /\ 1:r0=0)
```

One function per CPU, and `exists`: the outcome we're asking about.

`Result: Never` is what the memory model says. Value-returning atomics are fully ordered, so both loads can't return 0.

Tests from the kernel tree work unchanged: 22 of the 35 in

`tools/memory-model/litmus-tests/` convert today. The other 13 use spin locks, RCU or `smp_rmb()` / `smp_wmb()`, which blitmus doesn't support yet.



Step 2: generate the BPF programs

```
$ ./litmus2bpf.py litmus_tests/SB+fetchadd.litmus
Generated bss test: sb_fetchadd
$ ls .output/gen/sb_fetchadd*
.output/gen/sb_fetchadd.bpf.c
.output/gen/sb_fetchadd.c

$ cat .output/gen/sb_fetchadd.bpf.c
...
SEC("raw_tp/test_prog1")                /* P0 */
int handle_tp1(void *ctx)
{
    ...
    bpf_sense_barrier(&local_sense, num_threads);
    ...
    bpf_for (i, 0, BLITMUS_SLOTS) {
        barrier_wait(0, i);
        ...
        atomic_fetch_add(&shared.x[i], 1);
        r0 = READ_ONCE(shared.y[i]);
        ...
        shared.out_p0_r0[i] = r0;
    }
    ...
}
```

`sb_fetchadd.bpf.c`: one `raw_tp` program per process. The litmus body is copied as written, and every variable becomes an array of 10,000 slots, so one call runs the test 10,000 times.

`bpf_sense_barrier()` releases the CPUs together; `barrier_wait()` lines them up on each slot. Each slot's registers land in `out_*[i]`.

`sb_fetchadd.c`: the userspace harness that runs the programs and checks the results.



Step 3: build it

```
$ make bin/sb_fetchadd
BPF      .output/sb_fetchadd.bpf.o
GEN-SKEL .output/sb_fetchadd.skel.h
CC        .output/sb_fetchadd.o
MKDIR     bin
BINARY    bin/sb_fetchadd
```

BPF is your clang compiling the BPF side (`-mcpu=v4 -O2 -target bpf`): the compiler under test. Point it at any compiler with `make CLANG=clang-19 ...`. **GEN-SKEL** is bpftool's skeleton, **CC** the harness. The litmus body of P0 in the object:

```
$ llvm-objdump -d -j raw_tp/test_prog1 .output/sb_fetchadd.bpf.o
60:  r1 = 0x1
61:  r1 = atomic_fetch_add((u64 *) (r2 + 0x0), r1)      # atomic_fetch_add(&x[i], 1)
62:  r1 = r2
63:  r1 += 0x13880                                       # &y[i]
64:  r1 = *(u64 *) (r1 + 0x0)                           # READ_ONCE(y[i])
```



Step 4: load it: the verifier and the JIT

When `bin/sb_fetchadd` starts, libbpf loads both programs through the verifier, and the running kernel's JIT compiles them. Insn 61 on powerpc, from `bpf_jit_emit_atomic_ops()` (register moves omitted):

```
li      tmp1, 0
+ sync
ldarx   tmp2, tmp1, dst
add     tmp2, tmp2, src
stdcx.  tmp2, tmp1, dst
bne     <ldarx>
+ sync
```

The two `sync` s are what b1e7cee96127 added. Without them the RMW is relaxed, and the atomic on `x` can pass the load of `y`.

On arm64 insn 61 is `ldaddal`; on x86 it's `lock_xadd`.

See it on your machine while the test runs:

```
sudo bpftool prog dump jited name handle_tp1
```



Step 5: run it

```
$ sudo ./bin/sb_fetchadd
Starting litmus test with configuration:
  Test: SB+fetchadd
  Iterations: 4100
```

```
/* both condensed from .output/gen/sb_fetchadd.c */
for (it = 0; it < iterations; it++) {
    reset_state(skel);          /* zero .bss */
    pick_random_cpus(cpuses);
    for (p = 0; p < THREADS; p++)
        pthread_create(&t[p], NULL,
                       worker, &args[p]);
    for (p = 0; p < THREADS; p++)
        pthread_join(t[p], NULL);
    for (c = 0; c < BLITMUS_SLOTS; c++)
        check_cond(skel, &matches,
                   &non_matches, c);
}
```

```
static void *worker(void *arg)
{
    struct bpf_test_run_opts opts = {
        .sz      = sizeof(opts),
        .flags   = BPF_F_TEST_RUN_ON_CPU,
        .cpu     = a->cpu,
    };

    sched_setaffinity(0, sizeof(cpuset), &cpuset);
    pthread_barrier_wait(a->barrier);
    bpf_prog_test_run_opts(a->prog_fd, &opts);
}
```

Every round: random CPUs, one pinned thread per process, released together. The program runs on that CPU with preemption off: 10,000 slots per call, 4,100 rounds, 41M runs (`-i` changes it).



Step 6: read the verdict (powerpc)

JIT without the fix

```
Test SB+fetchadd Allowed
Histogram (4 states)
7085332  *>0:r0=0; 1:r0=0;
15981076 :>0:r0=0; 1:r0=1;
17519999 :>0:r0=1; 1:r0=0;
413593   :>0:r0=1; 1:r0=1;
Ok
Positive: 7085332, Negative: 33914668
Condition exists (0:r0=0 /\ 1:r0=0) is validated
```

JIT with b1e7cee96127

```
Test SB+fetchadd Allowed
Histogram (3 states)
8947286  :>0:r0=0; 1:r0=1;
9018695  :>0:r0=1; 1:r0=0;
23034019 :>0:r0=1; 1:r0=1;
No
Positive: 0, Negative: 41000000
Condition exists (0:r0=0 /\ 1:r0=0) is NOT validated
```

Each slot's registers go into the histogram; `*` marks the state `exists` asks about. With the fix it never shows up. Without it, it's 1 run in 6.

A `Result: Never` test with `Positive > 0` exits 1, so a script or CI can catch it.



Not just instructions: kfuncs

DEFAULT

bss

The test's variables live in ordinary global memory.

blitmus: arena

arena memory

The same test with its variables in an arena, reached through the JIT's arena access path.

blitmus: arena-recycle x

pages that come and go

x lives on a page that P0 gets from `bpf_arena_alloc_pages()` in every slot; `reset()` frees them all with `bpf_arena_free_pages()` before every round.

One line in the test's comment picks the backend. Bug 3 only shows up in the third: on the first two, the same message passing reads 0. 128 slots and 40,000 rounds keep pages recycling under the race.



What a 0 means, and what a hit means

A 0 can be a false negative

`BPF_PROG_TEST_RUN` works through the slots in order, and the per-slot rendezvous is a bounded spin, so the CPUs can drift apart and close the window.

So every family has a control that must show the weak outcome on the same machine: SB+add, MP+poonceonces, IRIW+poonceonces. If the control reads 0, the zeros next to it mean nothing.

A hit has to rule out the harness

For bug 3: bss and plain arena read 0, full barriers don't help, there's an independent reproducer, and the fix takes it from 25 to 0.

For bug 1: the same source file, machine and kernel go from 14.7% to 0 with a newer compiler. For bug 2: the only change is the two `sync` s, and they take it from 17.3% to 0.



SB is one shape. There are more.

MP

message passing

P0	P1
x = 1	r0 = y
y = 1	r1 = x

r0 = 1 but r1 = 0?

SB

store buffering

P0	P1
x = 1	y = 1
r0 = y	r1 = x

r0 = r1 = 0?

LB

load buffering

P0	P1
r0 = x	r1 = y
y = 1	x = 1

r0 = r1 = 1?

IRIW

two writers, two readers

P0, P1 write x, y
P2 reads x, then y
P3 reads y, then x

P2, P3 disagree?

WRC

a three-CPU chain

P0: x = 1
P1: sees x, y = 1
P2: sees y, reads x

x still 0?

The kernel tree ships 47 litmus tests built from these shapes. `diy7` from `herdtools` generates more: one command, up to 4 CPUs, 68 tests. Once `litmus2bpf.py` reads its output, the suite grows without anyone writing BPF.

What it covers today

test family	Result	arm64	x86
coherence: CoRR, CoRW, CoWR, CoWW	Never	✓ 0	✓ 0
SB, nothing ordered	Sometimes	46.3%	4.1%
SB, relaxed atomic	Sometimes	13.2%	not seen
SB with <code>smp_mb()</code> or fully ordered atomics	Never	✓ 0	✓ 0
MP, nothing ordered	Sometimes	7.3%	not seen
MP with release/acquire or an address dependency	Never	✓ 0	✓ 0
LB, nothing ordered	Sometimes	not seen	not seen
IRIW, nothing ordered	Sometimes	0.003%	not seen
3 and 4 CPU chains with ordering: ISA2, WRC, R, S, IRIW	Never	✓ 0	✓ 0
arena memory, ordered: MP+addr, SB+xchg	Never	✓ 0	✓ 0
arena pages recycled: MP+recycle	Never	✗ 25	✗ 11

`sudo ./run.sh` · 37 litmus tests · 41M runs each (5.12M for recycling) · arm64 built with clang trunk · "not seen": allowed, not observed

What this means for your BPF code

- **Say the ordering you mean.** Recent clang with `-mcpu=v4` compiles `__atomic_load_n(p, __ATOMIC_ACQUIRE)` and `__atomic_store_n(p, v, __ATOMIC_RELEASE)` to real `load_acquire / store_release` instructions (`__BPF_FEATURE_LOAD_ACQ_STORE_REL` says it can), and the kernel accepts them from v6.15
- **Build with LLVM 20+ or GCC 15+** if any code leans on an atomic as a barrier (bug 1)
- **Test where reordering is visible.** arm64 shows what x86 hides; bugs 1 and 2 were invisible on x86
- **Run blitmus** when you touch a JIT's atomics, a verifier rewrite of them, or a kfunc that manages memory



Next, and how you can help

Next

- a memory-model suite in `selftests/bpf`, run by BPF CI on x86_64, aarch64 and s390x; any `Never` outcome fails the run
- tests for what programs use:
`bpf_res_spin_lock()`, the ring buffer, the arena spin lock, load-acquire / store-release on every JIT
- generated families from `diy7`

Run it

```
git clone \
    https://github.com/puranjaymohan/blitmus
cd blitmus
git submodule update --init --recursive
make                # clang 20 or newer
sudo ./run.sh
```

Have riscv, s390 or loongarch hardware? Run it and send us the table.

Same litmus test, every layer, real hardware. If any layer drops the ordering, the histogram shows it.



Thank you

github.com/puranjaymohan/blitmus



Backup: herd7, klitmus7 and blitmus

	herd7	klitmus7	blitmus
input	litmus test	litmus test	litmus test
runs	nothing: enumerates executions against a model	a kernel module, one kthread per CPU	one BPF program per CPU
checks	the model	kernel C primitives + CPU	compiler, verifier, JIT, kfuncs + CPU

herd7 already has a BPF model (`bpf.cat`) and a BPF assembly front end. It answers whether hand-written bytecode is correct under the model; it never runs clang, the verifier, a JIT, a kfunc or a CPU. It would flag bug 1's `lock +=` if someone copied clang's output into a test by hand. Bugs 2 and 3 sit below the bytecode, where it can't look.



Backup: why BPF_PROG_TEST_RUN

- it runs a `raw_tp` program on a chosen CPU: with `BPF_F_TEST_RUN_ON_CPU` and the thread pinned there, `bpf_prog_test_run_raw_tp()` runs it in place with preemption disabled
- the harness controls exactly when each process starts, which real hooks don't give you
- it exercises the same verifier, JIT and kfuncs as any other program type; what it doesn't cover is attach-point context (XDP, struct_ops, tracing), which is future work



Backup: are the expected results right?

Every test's `Result:` line was checked with herd7 and the kernel memory model.

- 28 of 37 tests are valid LKMM input as written, and all 28 agree
- 6 atomics tests use blitmus's pointer-first `atomic_fetch_add(x, 1)` ; rewritten in LKMM form `( r1 = atomic_fetch_add(1, x) )`, all agree
- 3 dependency tests use `data[r0]` , which herd7's C front end doesn't accept

Making every test valid herd7 input as written is on the list, so the expected answer comes from the model rather than from a comment.

