BPF on arm64: Closing the Performance Gap

Linux Storage, Filesystem, MM & BPF Summit 2026

Puranjay Mohan <puranjay@kernel.org>

arm64 BPF JIT: Feature Parity Is Close

Feature	x86_64	arm64	Gap
BPF JIT	v3.0	v3.18	18 vers
Trampolines	v5.5	v6.0	5 vers
bpf_prog_pack	v5.18	v6.9	11 vers
BPF Exceptions	v6.7	v6.9	2 vers
Arena (PROBE_MEM32)	v6.9	v6.10	1 ver
Percpu Instructions	v6.10	v6.10	same
Private Stack	v6.13	v6.17	4 vers
Timed may_goto	v6.15	v6.18	3 vers
Arena Fault Reporting	v6.18	v6.18	same
Tracing Session	v7.0	v7.0	same
Stack Arguments	—	—	in dev *

On x86 only (not needed on arm64)

- **Direct tail call patching** — arm64 has no retpoline, so indirect BR/BLR has no penalty
- **BPF dispatcher** — binary-search dispatch to avoid indirect calls; unnecessary without retpoline

In progress

- **bpf_get_branch_snapshot()** — BRBE support for arm64 under review

(*) Stack arguments in development, landing simultaneously on both.

The gap went from 18 versions to zero. The question now is performance.

Why Is arm64 Slower? Five Architectural Differences

1. **Instruction cache coherency** — x86 hardware keeps icache coherent; arm64 needs explicit cache maintenance and IPIs to all CPUs
2. **Per-CPU atomics** — x86 can do CMPXCHG without LOCK; arm64 LSE atomics are always globally atomic
3. **READ_ONCE() under LTO** — every READ_ONCE becomes LDAR/LDAPR on arm64, adding acquire ordering to every volatile read
4. **Dependent cache misses** — pointer chases in `bpf_sk_storage_get()` serialize against cache miss latency
5. **Instruction selection** — the arm64 JIT uses instruction encodings that don't match what the hardware optimizes for

Instruction Cache Coherency

x86: hardware does it

Intel SDM §14.6:

> "A write to a memory location in a code segment that is currently cached in the processor causes the associated cache line (or lines) to be invalidated. [...] If the write affects a prefetched instruction, the prefetch queue is invalidated."

Stores to code automatically invalidate icache + prefetch queue on all cores. No software involvement.

arm64: software has to do it

ARM ARM B2.2.5 — three steps:

Step 1: No PE executing the modified instruction.

Step 2: Cache maintenance (broadcast):

```
DC   CVAU, Xn   ; Clean dcache to PoU
DSB  ISH         ; Barrier
IC   IVAU, Xn   ; Invalidate icache
DSB  ISH         ; Barrier
```

Step 3: Each PE must execute an ISB or context synchronization event.

Kernel cannot remotely execute ISB → **IPIs**.

What This Means for the BPF JIT

x86: write code, done

```
bpf_jit_binary_pack_finalize()
 → text_poke_copy()
   → __text_poke()
     1. Switch to temp mm
     2. memcpy(code)
     3. Switch back
```

No cache maintenance. No IPIs.

Hardware handles everything.

arm64: IPI every CPU

```
bpf_jit_binary_pack_finalize()
 → __text_poke()
   1. Write code via fixmap
   2. flush_icache_range()
     → caches_clean_inval_pou()
     → kick_all_cpus_sync()
```

```
void kick_all_cpus_sync(void) {
    smp_mb();
    smp_call_function(do_nothing, NULL, 1);
}
```

`do_nothing` is literally empty — the interrupt entry itself is the context synchronization event.

72-core server = 71 synchronous IPIs every time we load a BPF program.

Per-CPU Atomics: x86 vs arm64

BPF trampoline recursion detection calls `this_cpu_inc_return` on a per-CPU counter on every trampoline entry — that's an atomic on the hot path.

x86: per-CPU atomics are cheap

```
/* arch/x86/include/asm/percpu.h */

/* this_cpu_xchg() is implemented using
 * CMPXCHG without a LOCK prefix. */

/* CMPXCHG has no such implied lock
 * semantics as a result it is much more
 * efficient for CPU-local operations. */
```

`CMPXCHG` without `LOCK` is a local operation — no bus traffic, completes in L1.

Even with `LOCK`, Intel SDM §11.1.4 says the processor uses **cache locking**: it modifies the line internally without asserting LOCK# on the bus.

arm64: no equivalent

- LSE atomics (`CAS`, `SWP`, `LDADD`, ...) are **always globally atomic** across all cores in the shareability domain
- No local-only variant
- Operation travels to L2/interconnect even for per-CPU data no other core will ever touch

Removing Atomics from BPF Recursion Detection

```
/* Before: atomic, globally visible on every BPF call */
return this_cpu_inc_return(*(prog->active)) == 1;

/* After: non-atomic per-CPU u8[4], one slot per context */
u8 *active = this_cpu_ptr(prog->active);
active[rctx]++;               /* {NMI, hardirq, softirq, normal} */
val = le32_to_cpu(*(__le32 *)active);
if (val != BIT(rctx * 8))     /* any other slot changed? recursion */
    return false;
```

From {0,0,0,0}: normal context → {0,0,0,1}. NMI arrives → {1,0,0,1}.

NMI checks against 0x00000001, sees 0x01000001 → recursion detected.

25% improvement in fentry microbenchmark on Neoverse V2.

Safe because BPF programs run with migration disabled — the per-CPU data cannot be accessed by another CPU. arm64-only — on x86, per-CPU atomics are already cheap, so the u8 array logic degrades performance.

READ_ONCE() Becomes LDAR Under LTO

Normal: volatile load → LDR

```
#define __READ_ONCE(x) \
  (*(const volatile typeof(x) *)&(x))
```

With CONFIG_LTO: acquire load → LDAPR/LDAR

```
/* arch/arm64/include/asm/rwonce.h */

/* When building with LTO, there is an
 * increased risk of the compiler converting
 * an address dependency headed by a
 * READ_ONCE() into a control dependency
 * and consequently allowing for harmful
 * reordering by the CPU. */

#define __LOAD_RCPC(sfx, regs...)         \
  ALTERNATIVE(                             \
    "ldar"  #sfx "\t" #regs,  /* RCsc */  \
    ".arch_extension rcpc\n"               \
    "ldapr" #sfx "\t" #regs,  /* RCpc */  \
  ARM64_HAS_LDAPR)
```

Instruction	Ordering constraint
LDR	None — CPU reorders freely
LDAPR	Later accesses cannot pass it
LDAR	Same, plus Store-Release ordering

The cost is not in the load itself — L1 hit latency is similar. It's the ordering constraint: LDAR/LDAPR prevent the CPU from reordering later memory accesses past the acquire, which limits how much ILP the out-of-order engine can extract.

Every `READ_ONCE()` in the kernel pays this cost when LTO is enabled on arm64.

Dependent Cache Misses: bpf_sk_storage_get()

Two dependent cache misses in the fast path:

```
/* sk->sk_bpf_storage is fetched first (sk is hot, usually free) */

/* cache miss #1: fetch sdata from local_storage */
sdata = rcu_dereference(local_storage->cache[smap->cache_idx]);

/* cache miss #2: fetch sdata->smap (dependent on miss #1) */
if (sdata && rcu_access_pointer(sdata->smap) == smap)
        return sdata;
```

`perf annotate` on arm64 confirms — cycles concentrate on these two loads:

```
      ldar    x9, [x8]          ; load sdata ptr from cache
28.16 cbz     x9, ...           ; MISS #1: local_storage is cold

      ldar    x8, [x9]          ; load sdata->smap (depends on x9)
23.18 cmp     x8, x0            ; MISS #2: sdata is a separate kmalloc obj
```

~51% of cycles in this function waiting on two dependent cache misses. Out-of-order execution cannot help — each load depends on the previous result.

Martin KaFai Lau has patches to inline local storage data into the socket itself.

We Need Real Workload Benchmarks

The existing BPF bench tests measure individual operations in isolation — a map lookup here, a kprobe there.

Real BPF programs combine parsing, map lookups, branching, and encapsulation in a single call chain. What matters is how these interact: register pressure, spill patterns, inlining decisions, branch layout.

Microbenchmarks miss these interactions entirely.

Approach: workload benchmarks in selftests

- First: **XDP load balancer** based on Katran
- Next: **sched_ext schedulers**, tracing programs
- Goal: a suite that exercises BPF the way it's used in production

Why now

- The JIT peephole optimizations show 25-31% per-instruction gains in microbenchmarks — but does that translate to real workloads?
- Register utilization experiments need stable numbers with low variance
- arm64 vs x86 comparison needs a shared baseline

XDP Load Balancer Benchmark (based on Katran)

L4 load balancer datapath: parsing → VIP lookup → per-CPU LRU with CH fallback → server selection → stats → IPIP encap.

Single-flow baseline	p50	stddev	p99
tcp-v4-lru-hit	74.30	0.08	74.48
tcp-v4-ch	101.73	0.11	102.01
tcp-v6-lru-hit	76.77	0.14	77.04
udp-v4-lru-hit	107.42	0.22	107.90
Diverse flows (4K src addrs)	**p50**	**stddev**	**p99**
tcp-v4-lru-diverse	86.63	0.37	89.04
LRU stress	**p50**	**stddev**	**p99**
tcp-v4-lru-miss	440.39	35.75	550.62
tcp-v4-lru-warmup	317.75	9.55	356.20
Early exits	**p50**	**stddev**	**p99**
pass-non-ip	5.71	0.03	5.76

Timing inside the BPF program via `bpf_ktime_get_ns()`. 24 scenarios covering the full code-path matrix. Auto-calibrated batch sizes. Each scenario validates correctness before benchmarking.

Reading the CPU Optimization Guide Pays Off

I went through the Neoverse V1, V2, N2 Software Optimization Guides and V2 Core TRM looking for instruction sequences where the JIT can do better.

Example: register MOV

```
JIT emitted: ADD Rd, Rn, #0
  → ALU pipeline (1 cycle)

Should emit: ORR Xd, XZR, Xn
  → register renaming (0 cycles)
```

Neoverse V2 eliminates ORR-based MOVs in the rename stage — they never reach the execution units. ADD #0 must go through the ALU.

MOV encoding	Pipeline	cycles/op
`ORR Xd, XZR, Xn`	register renaming	**0.750**
`ADD Xd, Xn, #0`	ALU	1.005

~25% faster in a MOV-only microbenchmark. Merged.

JIT Peephole Optimizations

All benchmarked on Neoverse V2 (cycles per operation):

Optimization	Before	After	Improvement
MOV: ORR vs ADD #0	1.006	0.751	**25.4%**
CMP imm vs MOV+CMP	0.510	0.351	**31.2%**
STP offset vs pre-index	1.009	0.724	**28.2%**
LDR imm vs MOV+LDR	0.347	0.337	2.9%
CBZ vs CMP+B.eq	0.528	0.519	1.7%
TBNZ vs TST+B.ne	0.528	0.519	1.7%

Combined, these reduce JIT code size by ~2.7% across Meta internal BPF programs.

ORR-based MOV is merged. For the rest, I want to finalize the XDP benchmark first so we can measure end-to-end impact before sending upstream.

BPF to arm64 Register Mapping

arm64	BPF	Role
x0	R1	arg 1
x1	R2	arg 2
x2	R3	arg 3
x3	R4	arg 4
x4	R5	arg 5
x5	—	**free** (caller-saved)
x6	—	**free** (caller-saved)
x7	R0	return value
x8	—	**free** (caller-saved)
x9	AX	temp
x10	TMP1	JIT temp
x11	TMP2	JIT temp
x12	TMP3	JIT temp
x13-18	—	**free** (caller-saved)

arm64	BPF	Role
x19	R6	callee-saved
x20	R7	callee-saved
x21	R8	callee-saved
x22	R9	callee-saved
x23	—	**free** (callee-saved)
x24	—	**free** (callee-saved)
x25	FP (R10)	frame pointer
x26	tcc_ptr	tail call count
x27	priv_sp	**free** if no priv stack
x28	arena_vm	**free** if no arena
x29	FP	arm64 frame pointer
x30	LR	link register
SP	—	stack pointer

Between BPF registers, JIT temporaries, and special-purpose registers, about 20 of arm64's 31 GPRs are spoken for. The rest — x5, x6, x8, x13-x18, x23, x24, and conditionally x27, x28 — are free and available for argument passing and spill/fill caching.

Stack Arguments: Using Free Registers

BPF is limited to 5 arguments (R1-R5). kfuncs and subprogs increasingly need more.

Joint work with Yonghong Song: remap R0 from x7 → x8, which frees x5-x7 for arguments 6-8.

arm64	Current mapping	New mapping
x0-x4	R1-R5 (args)	R1-R5 (args)
x5	**free**	**arg 6** (in register)
x6	**free**	**arg 7** (in register)
x7	R0 (retval)	**arg 8** (in register)
x8	**free**	**R0 (retval, remapped)**

x8 is AAPCS64's indirect result register — caller-saved, not used for argument passing. Arguments 9-12 spill to stack at `[SP+0..+24]`.

AAPCS64-compliant. Up to 12 arguments. In development — arm64 and x86 landing in parallel.

Experiment: Register Utilization for Spill/Fill

The idea

x23, x24 are always free. x27, x28 are free when private stack / arena aren't used. Meanwhile, the compiler is spilling and filling through the stack constantly.

What if we cache hot stack slots in these free registers?

```
Before:
  STR X_bpf, [FP, #-off]  ; spill (memory)
  LDR X_bpf, [FP, #-off]  ; fill  (memory)

After:
  MOV X23, X_bpf           ; spill (0-cycle)
  MOV X_bpf, X23           ; fill  (0-cycle)
```

ORR-based MOV is eliminated by register renaming on Neoverse V2 — spill/fill becomes free.

Early results

- Katran benchmark: **2-3.5% PPS improvement** (needs the XDP benchmark for proper measurement)

What it would need

- **LLVM** — Eduard's r11 branch (BPF_REG_SB)
- **Verifier** — precision propagation across spill base access
- **JIT** — map stack slots to free registers

What We Covered

arm64 BPF JIT is close to feature parity with x86. Recent features land on both architectures simultaneously.

Performance gaps traced to five architectural differences:

- **Icache coherency** — 71 IPIs per JIT load on a 72-core server
- **Per-CPU atomics** — LSE atomics are globally atomic; removed them from recursion detection, 25% fentry improvement
- **READ_ONCE + LTO** — every volatile load becomes an acquire on arm64
- **Dependent cache misses** — `bpf_sk_storage_get()` serializes against miss latency
- **Instruction selection** — 25-31% per-instruction improvement from reading the Neoverse optimization guides

What's In Flight

- 5 JIT peephole patches — finalizing the XDP benchmark first to measure end-to-end impact
- Register utilization for spill/fill — early experiment, needs benchmark numbers and LLVM/verifier work
- XDP load balancer benchmark — sent to bpf-next; sched_ext benchmark next
- `bpf_sk_storage_get()` inline patches — working with Martin
- Stack arguments — in development with Yonghong, arm64 and x86 in parallel

Thanks!

Puranjay Mohan <puranjay@kernel.org>