LINUX · GPU & ACCELERATOR OBSERVABILITY

Everyone Builds Their Own Pipe

The missing Linux standard for streaming performance data off non-CPU devices

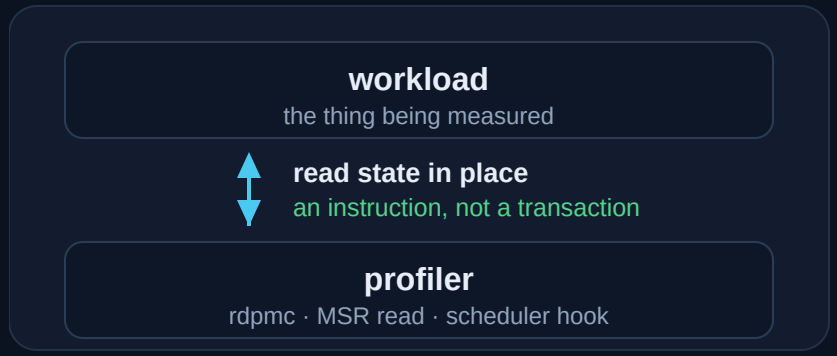
The missing UAPI for device-produced telemetry

THE PREMISE

You can read a CPU. A device has to send.

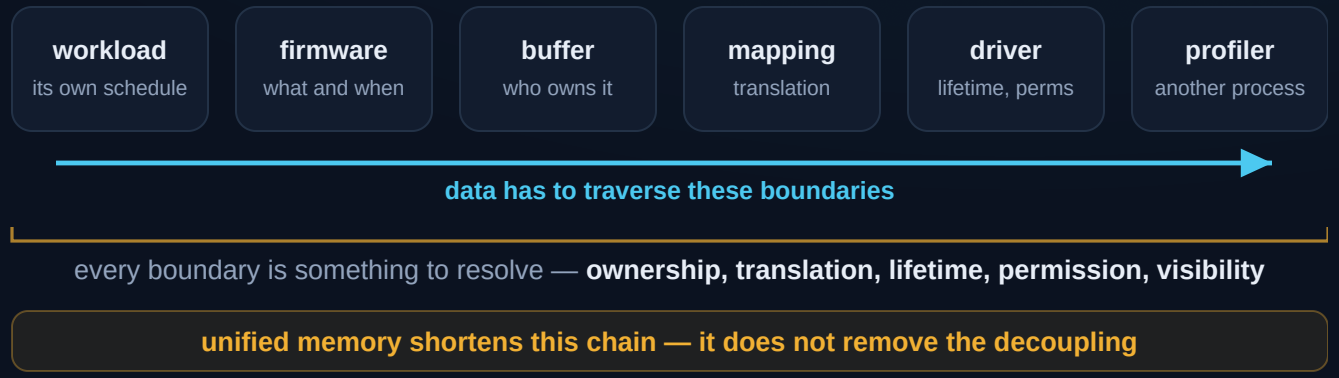
Direct access is what makes CPU profiling work. A device cannot offer it.

CPU — DIRECT ACCESS



same core · same clock · same address space — **no boundary crossed**

DEVICE — BOUNDARIES, NOT DISTANCE



every boundary is something to resolve — **ownership, translation, lifetime, permission, visibility**

unified memory shortens this chain — it does not remove the decoupling

Once collection has to be asynchronous — the device writes, you drain later — you are outside everything perf gives you for free.

GETTING THE DATA OUT

Three ways off the device

Each mode pays a different price — and two of them change the thing you are measuring.

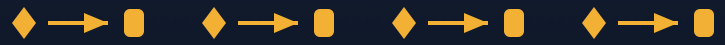
01 · POLLING

check, then read — most checks find nothing

2 round trips per item

02 · PUSH + SIGNAL

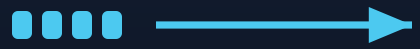
the device signals data landed, then you read



1 round trip + signalling

03 · FIRMWARE BATCHING

accumulate, then hand over many at once



1 round trip per N items

Costs shown are for a discrete device. On unified memory the traversal is cheaper — **the ordering and cooperation costs are unchanged**.

THE HIDDEN COST OF 01 AND 02

Per-item collection usually needs the **application's cooperation** — a profiler injecting counter read/write packets into its queue. The workload being measured is no longer the workload that shipped. Only batching leaves the application untouched.

THE OBVIOUS ANSWER

perf already solved this — for CPUs

The **AUX ring buffer** (Linux 4.1) exists precisely so hardware can write a trace stream straight into userspace-mapped pages.

It is a real transport

Hardware-written pages mapped to userspace. Head/tail framing, `poll()`, watermark wakeups, explicit truncation accounting, and an auth model the kernel has already argued about for a decade.

Nobody has to reinvent any of that on the CPU side.

And it carries serious volume

Intel PT — `pt.c` DMAs a compressed control-flow stream into the AUX area at hundreds of MB/s per core.

ARM CoreSight ETM / SPE — a source→link→sink fabric funnelled into the same ring.

This class of problem is solved, and solved once, for the CPU. So why does every device vendor still build their own?

GAP ONE — ADDRESSING

There is no device in the tuple

`perf_event_open(attr, pid, cpu)` — every event is anchored to a **task**, a **CPU**, or both.

The (pid, cpu) matrix

PID	CPU	MEANING
≥0	−1	follow this task everywhere
≥0	≥0	this task, on that CPU
−1	≥0	all tasks on that CPU
−1	−1	invalid

There is no argument that names **a device**, and none that names a scope on one.

- Device and uncore PMUs do exist — Intel and AMD uncore, RAPL, CXL, ARM CMN. They are system-wide counters, not a per-client stream.
- They answer "**what did the device do**". They cannot be opened against a device and a scope on it, because there is nowhere to say it.
- Every vendor invented a private way to say it. That is the first thing they all rebuilt.

GAP TWO — THE BUFFER

The AUX buffer belongs to the event

Its producer is a CPU. That assumption is baked into who allocates the memory and when it goes away.

Kernel-allocated, per event

AUX pages are allocated by perf for one event and torn down with it. The consumer never chooses the memory — and a device cannot be handed pages it was never mapped to write.

Producer tied to a CPU

PT and CoreSight trace **the core the scheduler is switching**. The producer *is* the thing being context-switched, which is exactly why the per-task model could be made to fit.

Lifetime ends with the task

A device keeps DMA-writing after the submitting thread is descheduled or gone. The buffer has to outlive the event that opened it — AUX has no way to express that.

perf is not inadequate. It is CPU-shaped — no way to name a device, and a buffer model that assumes the producer is a core.

THE CONTRACT

What a device transport must define

perf gives a CPU transport all of this for free. A device transport must state it.

Buffer ownership

Kernel-allocated and mapped out to userspace, or process-allocated at a device VA the kernel resolves? Different lifetime rules, different failure modes — and every vendor picked one.

Lifetime across three parties

The device is DMA-writing, the driver holds a mapping, the consumer holds an fd. Any of them can go away first. **Who pins what, and until when?**

Visibility

When is a record complete and safe to read? Write ordering and release semantics need an actual rule. Coherent memory makes this **cheap, not automatic**.

Overrun

The producer laps the consumer. Counted, signalled, or silent? A trustworthy loss count is the difference between usable data and **a guess**.

Wakeup

How does the consumer learn there is data — a head pointer it polls, `poll()`, a doorbell, a watermark? Cheap-and-laggy and low-latency are both legitimate. Offer both.

Revocation

Device reset, FLR, hot-unplug, or the target exits mid-stream with DMA in flight. What the consumer sees then has to be **a contract, not an accident**.

None of this is exotic. All of it is solved today — six times over, privately, behind six incompatible ioctls.

THE CONSEQUENCE

So every vendor reinvents the same wheel

Every vendor builds a private ioctl interface from scratch. Most converge on the same shape — and none of them interoperate.

VENDOR / DRIVER	INTERFACE
Intel i915	`DRM_IOCTL_I915_PERF_OPEN` — ioctl → stream fd → `read()` OA reports, per-context. ~5,300 LOC
Intel Xe	`DRM_IOCTL_XE_OBSERVATION` — same shape, rebuilt from scratch, not wire-compatible with i915
AMD ROCm / KFD	`/dev/kfd` ioctls + AQL packet injection into the submission stream
NVIDIA	CUPTI userspace library → private `/dev/nvidia*` ioctls
ARM Mali — Panfrost	`PERFCNT_ENABLE` / `_DUMP` — synchronous dump, whole-device, one user at a time
ARM Mali — Panthor	`DRM_PANTHOR_PERF_CONTROL` — streaming ring, insert/extract indices, eventfd

Even the new `drivers/accel` subsystem standardized device **nodes** — and says nothing about how diagnostic data gets off the device.

INTEL ALREADY LIVED THIS

> ***"The first prototype of this driver was based on the core perf infrastructure … we found we were breaking or working around too many assumptions baked into perf's currently cpu-centric design.***
>
> ***perf_event_open() requires events to either relate to a pid or a specific cpu core, while our device pmu related to neither."***
>
> — kerneldoc, drivers/gpu/drm/i915/i915_perf.c · "History and Comparison with Core Perf"

They tried the standard path. They abandoned it. Then wrote **~5,300 lines** of bespoke DRM ioctl to work around a gap the kernel never closed — and the next Intel driver rewrote it **again**.

"JUST USE AN EXISTING PRIMITIVE"

No single primitive solves all three problems

PRIMITIVE	DEVICE-WRITABLE BUFFER	FRAMING + POLL + OVERRUN	CROSS-PROCESS HANDOFF
perf AUX ring	no — per-event, dies with it	yes	ptrace / CAP_PERFMON
dma-buf	yes — that is the point	no framing / no notify	yes — fd possession
anon_inode fd	plumbing only	you build it yourself	yes — fd
eBPF ring buffer	no — CPU writes it	yes	fd, CPU-owned
relay / relayfs	no — kernel writes it	yes (both modes)	debugfs perms
LTTng-UST / CTF never merged upstream	no — app/CPU emits	yes	daemon-mediated

Each solves a piece. None combines them — and none of them can even name the device you want the stream from.

THE ASK

There should be a perf for the device

One merged way to carry device-produced diagnostic data to userspace — counters, traces, samples, whatever the device emits.

> **`perf_event_open()` takes (what, pid, cpu).**
> **A device interface needs (what, device, scope).**

01
Name the device
Address a device, and a scope on it, at open time. Nothing else is reachable until you can say which.

02
Device-writable buffer
A DMA target whose lifetime is independent of the task that requested it — the dma-buf strength.

03
Streaming protocol
Head/tail, poll *and* IRQ, counted overrun — the perf and relay strength.

04
Cross-process handoff
An fd-possession delegation from producer to profiler — solved once, not per vendor.

How that handle is *obtained* — an existing client id, a device node, something new — is an implementation question. And what identity the records carry stays vendor-specific. **This is about the pipe, not the payload.**

The fragmentation isn't a vendor failing. It's a missing kernel primitive.

DESIGNING THE INTERFACE

Open Problems

Agreeing there is a gap is the easy part. None of what follows is settled.

01

The entry point

Give `perf_event_open()` a device axis, add a new syscall, or generalize at the DRM/accel layer? perf brings tooling and an auth model but is deeply CPU-centric; a DRM interface leaves out NPUs and DPUs. And discovery — how does a tool ask what a device can even produce?

02

A bidirectional path

Today's interfaces are read-mostly. Real use needs start/stop mid-run, reconfiguring which routines collect, arming triggers — commands that reach **firmware**, not just the driver. And writing to a device is a bigger privilege than reading from it.

03

Buffer shape & record format

Ring? Double, triple, N-buffer? Chunked with interrupts? Hardware differs sharply — even between GPUs. The device needs to **declare** its buffer topology and its record layout rather than be forced into one.

THE CENTRAL TENSION

Flexible enough for genuinely disparate hardware, without pushing a bespoke decoder into every tool. Total flexibility recreates the Intel PT problem — thousands of lines of producer-specific decode, and `perf` patched forever to keep up. A rigid kernel-imposed format fits nobody. The likely way out is **self-description**: the driver ships a machine-readable schema alongside the data. **BTF** already does exactly this for eBPF. **CTF** does it for traces — but LTTng was never merged upstream, so the tracer is not on the table even where the format idea is worth borrowing.

APPENDIX

References

The perf model & the gap

`perf_event_open(2)` — man7.org

`kernel/events/core.c` — `perf_invalid_context` enforcement

Perf ring buffer / AUX — docs.kernel.org/userspace-api/perf_ring_buffer

Intel PT — `arch/x86/events/intel/pt.c`; LWN 629480

CoreSight — docs.kernel.org/trace/coresight

Vendor reinventions

`i915_perf.c` kerneldoc — the "related to neither" quote

Xe — `xe_oa.c`, `DRM_XE_OBSERVATION`

AMD KFD — `kfd_ioctl.h`; rocprofiler-sdk docs

NVIDIA CUPTI — docs.nvidia.com/cupti

DRM fdinfo — docs.kernel.org/gpu/drm-usage-stats

Panfrost `panfrost_perfcnt.c` · Panthor perf uAPI v3 (dri-devel, 2025)

`drivers/accel` intro

Kernel primitives

dma-buf — docs.kernel.org/driver-api/dma-buf · `anon_inodes.c` · BPF ringbuf — docs.ebpf.io · relay — docs.kernel.org/filesystems/relay · LTTng-UST / CTF — lttng.org

Describing the data

BTF — docs.kernel.org/bpf/btf · CTF / babeltrace — diamon.org/ctf · Intel PT decoder `tools/perf/util/intel-pt-decoder` · OpenCSD (CoreSight decode)