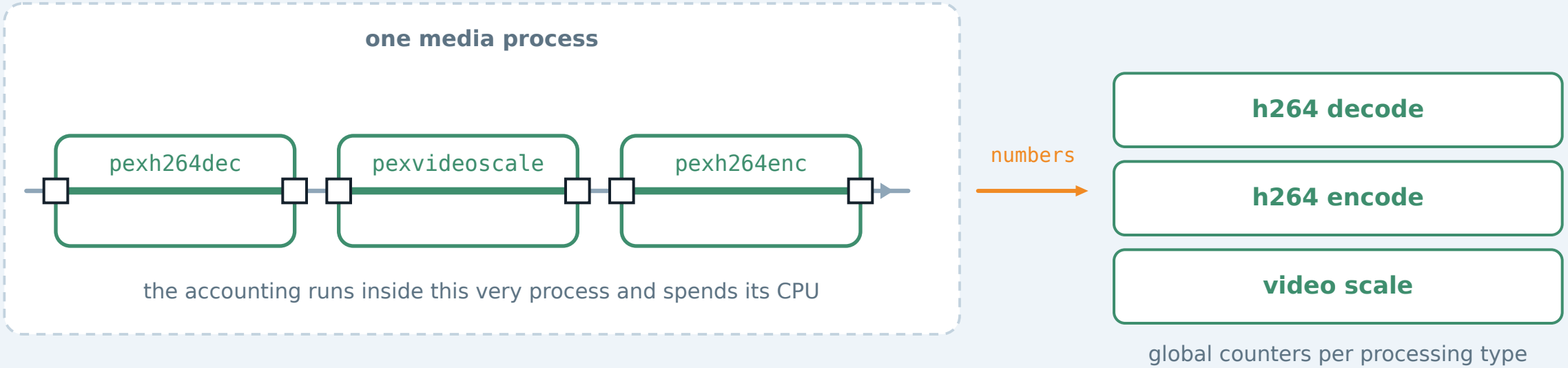
Always-On Tracing

Permanent CPU accounting in production with the GStreamer Tracing API

Sergei Kovalev · Pexip

GStreamer Conference 2026



always-on · in production

Pexip Infinity

A distributed video conferencing system: media nodes around the world, each participant joins the nearest one.



participant media: SIP, H.323, WebRTC

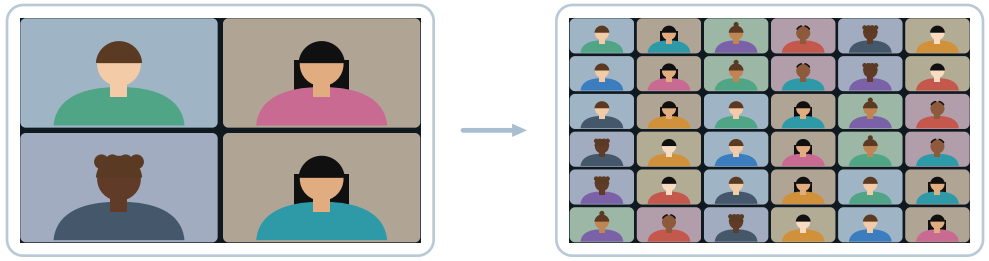
backplane between nodes

Participants come from different countries and different devices, each joining the nearest node. The nodes exchange already rendered media. Product details are on the internet; for this talk one part matters — the transcoding one, the part that burns CPU.

What changes the load during a call

Layouts, per-participant mixes and extra processing are switched on and off right in the middle of the conversation.

the layout decides how many streams go into the mix



2 × 2

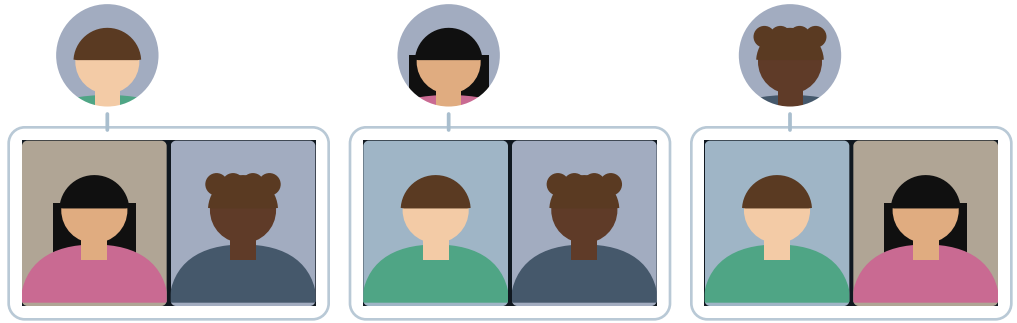
4 streams

6 × 6

36 streams

and the scalers work differently in each layout: different sizes, different amount of work

self-view is removed — so every participant has their own mix



mix 1 mix 2 mix 3

a participant never sees themselves in their own mix — so each one needs a separate encoder

and extra processing can be switched on at any moment

AI framing

segmentation

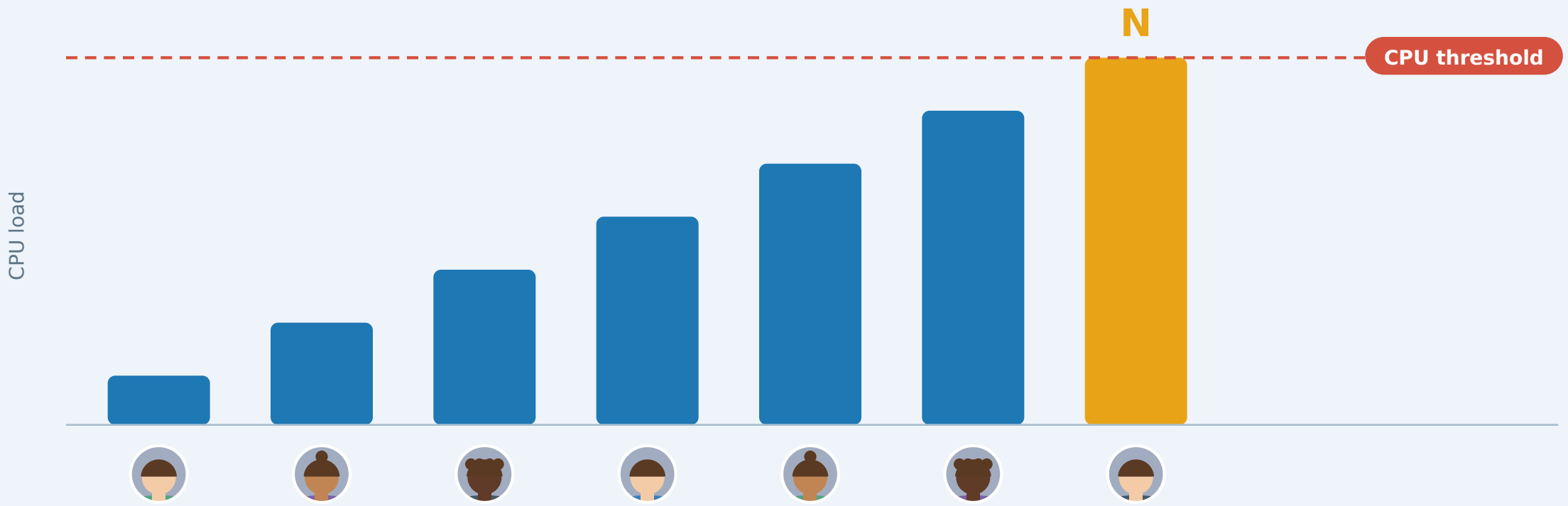
background blur

audio denoising

What will actually be running a minute from now is not known in advance. Hence the question: how do you plan such a load and spread it across nodes?

How this was solved in the early days

The question is not new: it came up at the dawn of Infinity. The system was simpler then — one main video codec and a few layouts.



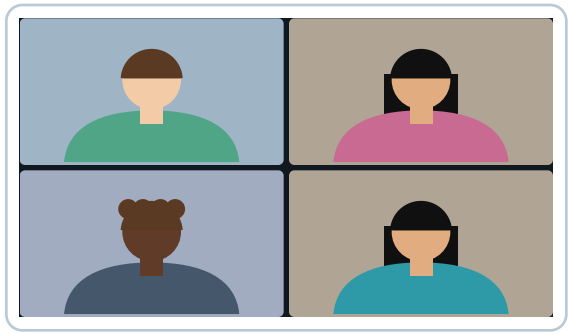
the benchmark adds simulated users one at a time

However many such users fit below the chosen threshold — that is what this hardware holds. The procedure runs once, right after the node comes up.

Infinity evolved — and the model stopped adding up

New codecs, layouts, AI and a new usage pattern. The old simple arithmetic accounted for none of it.

then

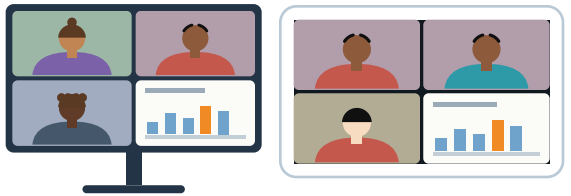


one main video codec
a few layouts

now



custom 6 × 6 layouts



meeting rooms with two screens:
two video streams instead of one

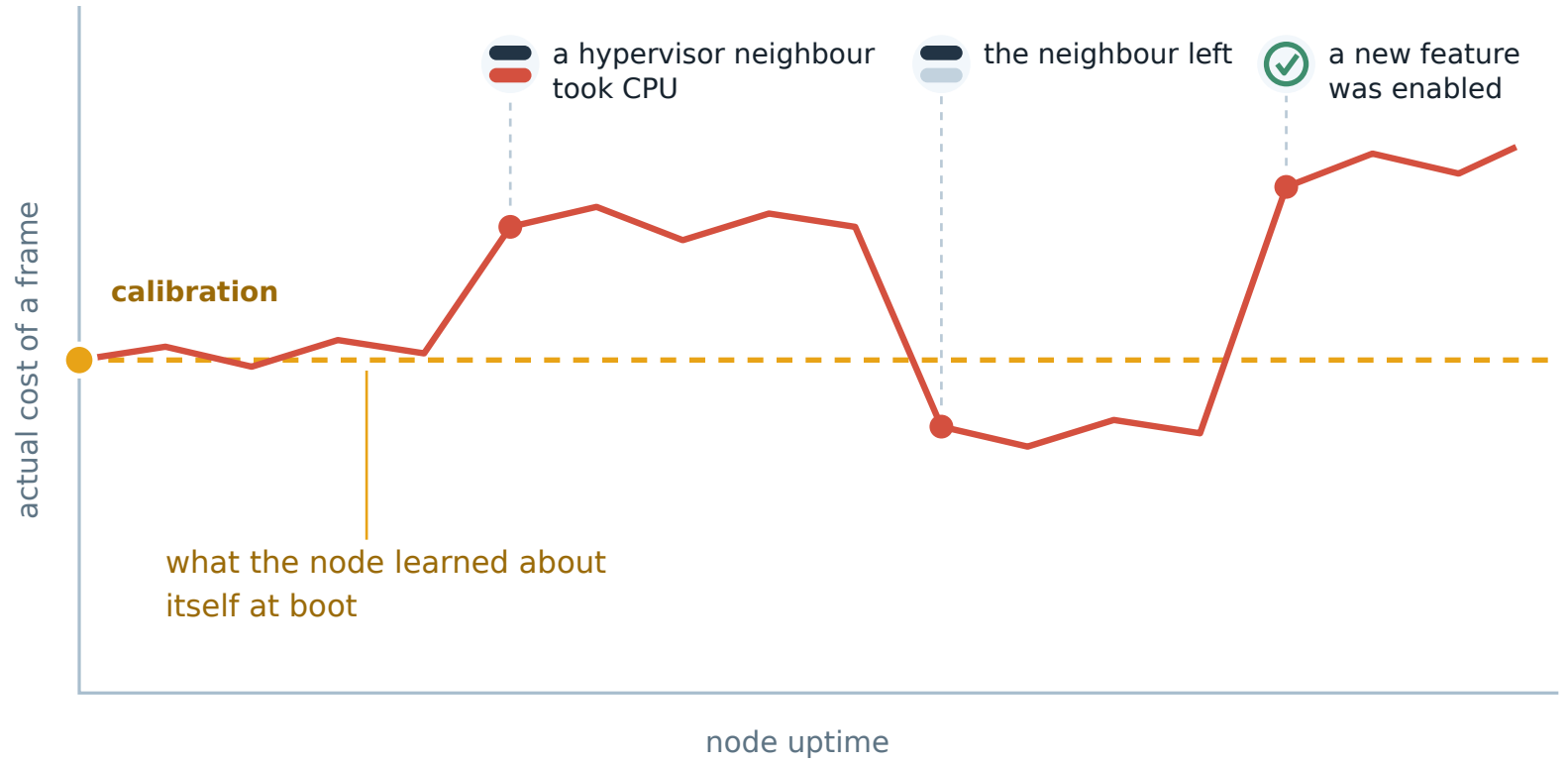
face detector · segmentation · audio denoising · new codecs · other resolutions

each of these features adds load of its own

A model built around one typical user started to be systematically wrong: two conferences with the same number of participants began to produce completely different load.

Boot-time calibration is a single point

After start-up the node runs a short test and gets an estimate of its hardware. There is a starting point. But only one.



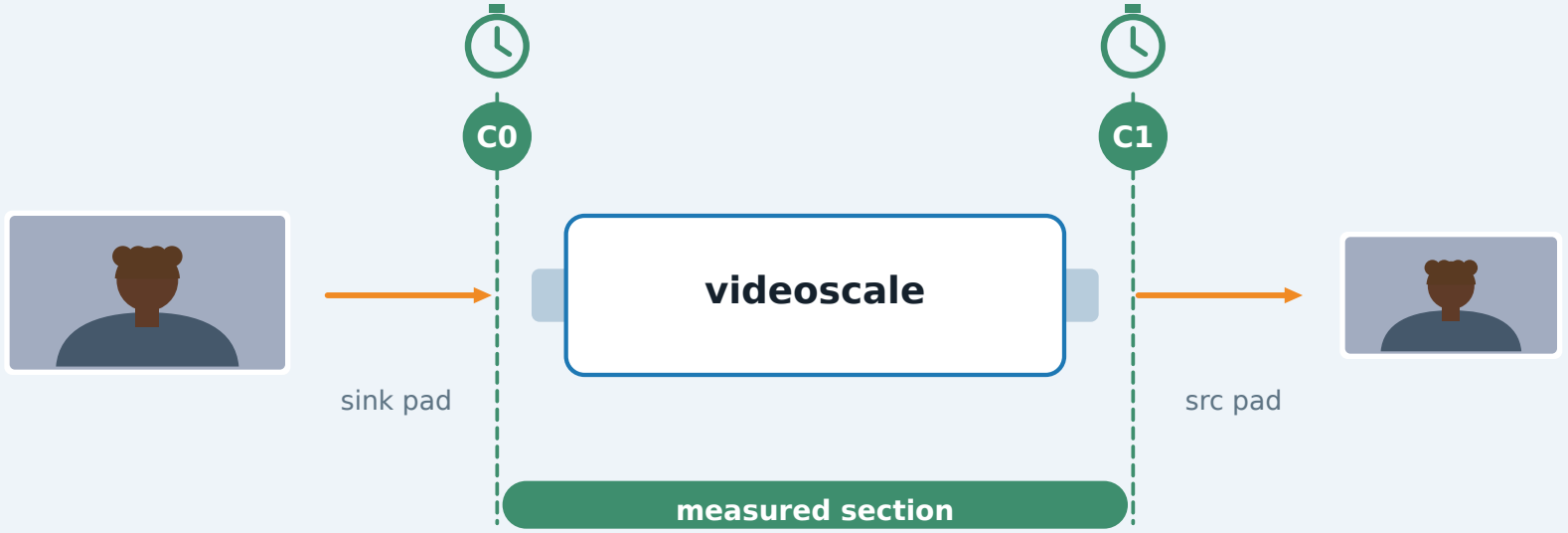
Calibration cannot test everything:

- a single encoder already has several modes, settings and resolutions
- plus other codecs, scalers, mixers, neural networks
- a full sweep would mean the machine runs benchmarks instead of calls

So we do not throw calibration away — we complement it with measurements in production: what encoding, decoding, scaling and mixing cost on this node right now.

The idea: measure CPU at the element boundaries

Sink pad is the input, src pad is the output. On the synchronous push path the work is done by the thread that brought the buffer.



C1 − C0

the difference between two readings of the same thread CPU-time counter — that is what we attribute to the element

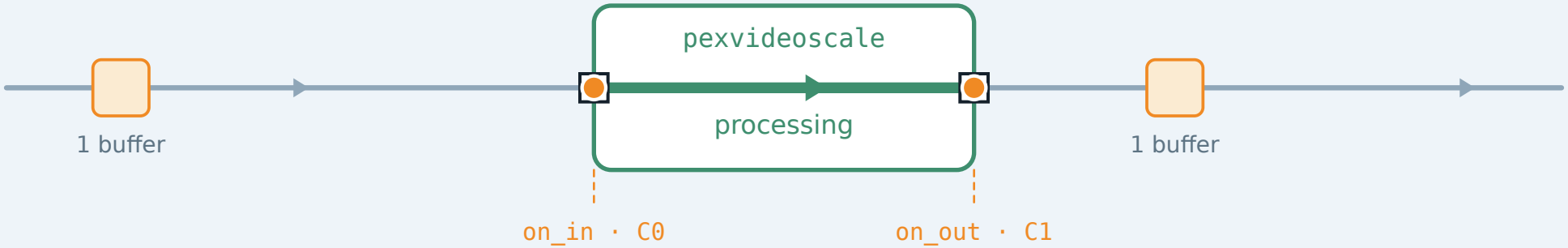
In Linux there is CLOCK_THREAD_CPUTIME_ID for exactly this — a cumulative CPU-time counter of the current thread. It is not a timer per element: we simply take two readings and subtract.

First implementation — buffer pad probes

A probe is a callback GStreamer invokes on a pad when a buffer passes through it.

```
gst_pad_add_probe (sinkpad, GST_PAD_PROBE_TYPE_BUFFER, on_in,  ...);
gst_pad_add_probe (srcpad,  GST_PAD_PROBE_TYPE_BUFFER, on_out, ...);

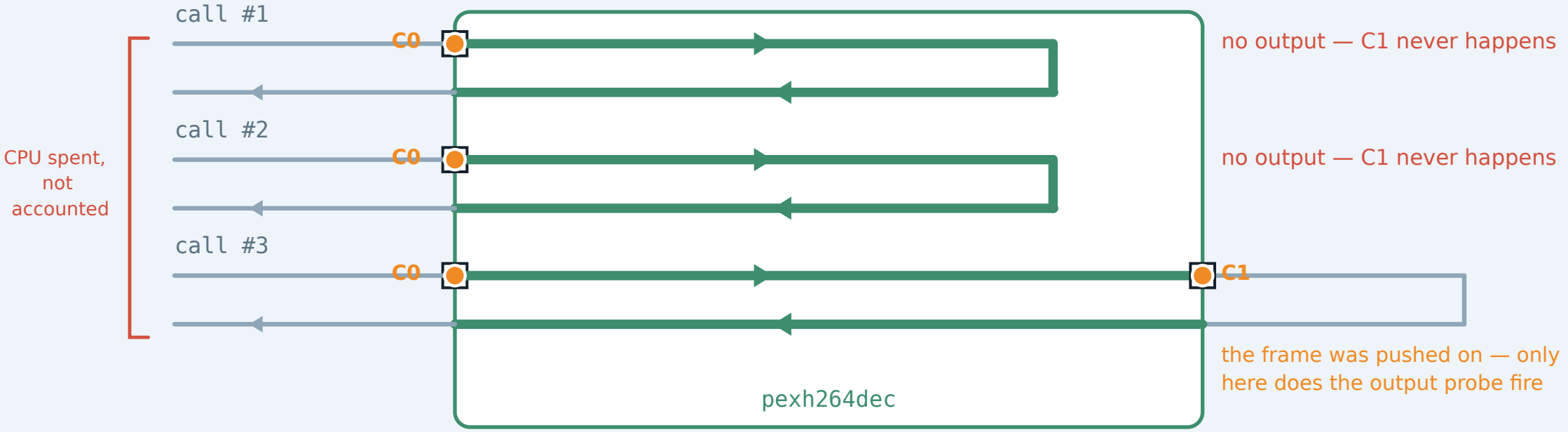
on_in  : C0 = cpu_now ();  in_bytes  += size;
on_out : cpu += cpu_now () - C0;  out_bytes += size;
```



On a simple element this works: one input buffer, one processing step, one output buffer. No plugin changes needed. But the output probe only means the buffer was pushed on — not that the element finished its work.

Where this breaks: the decoder

Three chain calls in a row; each time the thread enters the element, works and returns.



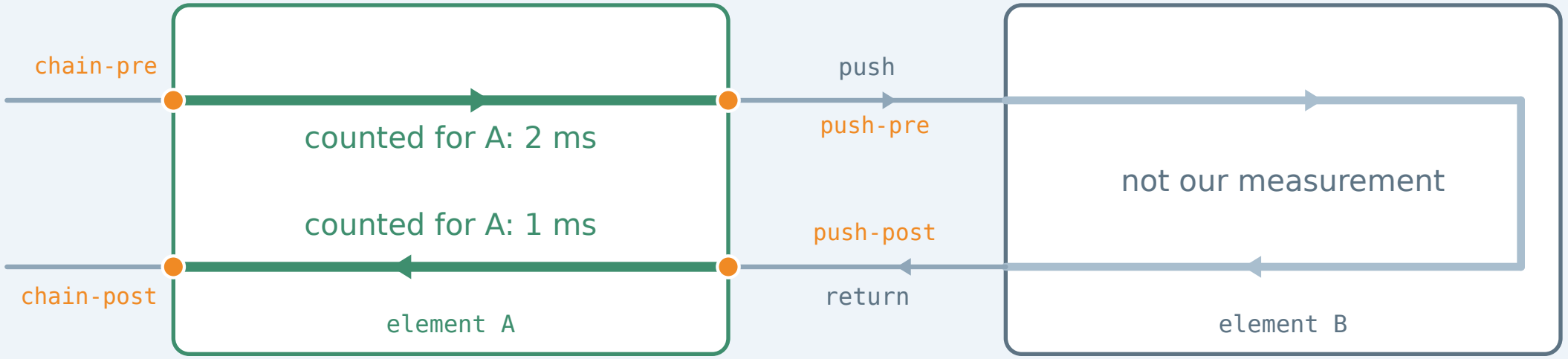
For a decoder, input and output do not match one to one: the first fragments produce no frame, so the output probe never fires for them. All the work of those calls was lost and got attributed to the last call instead.
The measurement has to be closed on the return from chain — and the pad-probe API has no such hook.

The fundamental gap in the pad-probe API: there is no point after the return from chain and from push.

The Tracing API gives the missing post points

One global GstTracer per process. The callback runs synchronously, on the very same thread.

```
gst_tracing_register_hook (tracer, "pad-chain-pre",  G_CALLBACK (chain_pre));
gst_tracing_register_hook (tracer, "pad-chain-post", G_CALLBACK (chain_post));
gst_tracing_register_hook (tracer, "pad-push-pre",   G_CALLBACK (push_pre));
gst_tracing_register_hook (tracer, "pad-push-post",  G_CALLBACK (push_post));
```



Between push-pre and push-post A has no open measurement, so B's work never lands in A's counter: A gets exactly 2 + 1 ms. This is not background trace-log reading and not GstBus.

But it is switched on for the whole process

The Tracing API is a global mechanism: its observation points are baked into GStreamer core all over the code.

```
GST_TRACER_DISPATCH (key, ...) {
  if (tracing_enabled) {            /* enabled globally  */
    l  = lookup (key);              /* our hook? no      */
    nl = lookup (NULL);             /* listener for all? */
    ...
}
```

where such points sit

- pad chain
- pad push
- element message
- object ref / unref
- mini object ref / unref
- query, event, link…

green — what we subscribe to

gst_object_ref / unref and points like them run this check on every call, even though we never subscribe to them.

So the cost of an always-on tracer cannot be judged by your own callback alone: the whole process pays.

Switching off unused categories at build time

A point disabled at compile time does no dispatch and does not evaluate its arguments — stronger than just not registering a callback.

```
-Dgstreamer:tracer_hooks_disabled=\
    refcount,lifecycle,memory,queries,events,topology
```

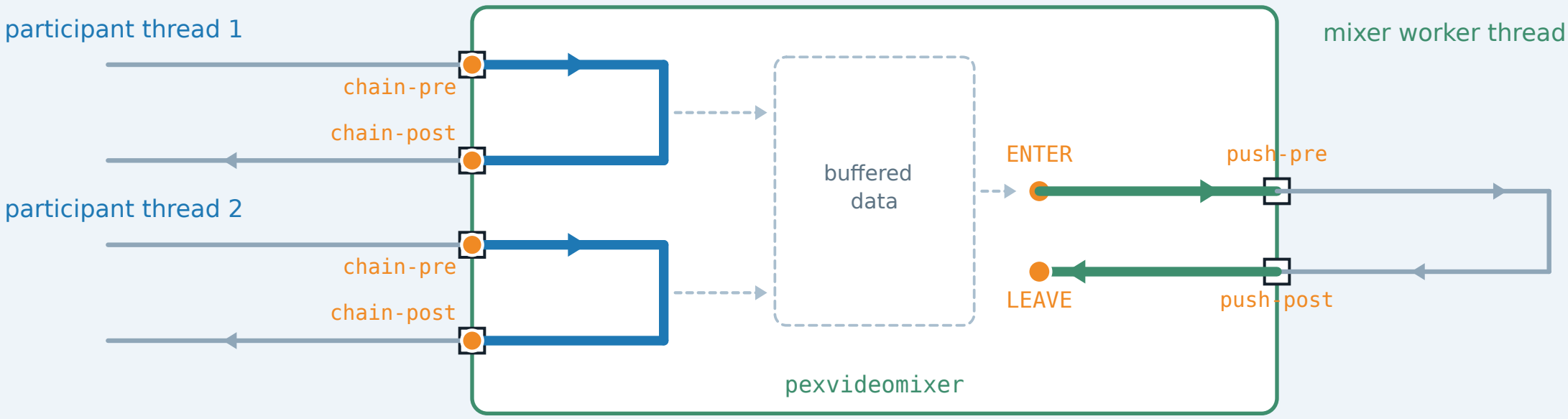
kept enabled

`scheduling`	pad-chain and pad-push: these are what cut execution into the sections we measure
`messages`	element messages: needed where an element creates a worker thread inside itself

A debug build keeps every category — it is needed for the leaks and latency tracers.

Why the messages category is needed

Some elements create a thread inside themselves: that worker thread has no incoming chain to start from.



> The worker thread's boundaries come from the STREAM_STATUS message — ENTER before the first result, LEAVE after the last one. We intercept it as it is posted, on the worker thread itself: GstBus would hand us another thread's clock. Source elements are exactly the same case: their own task inside, no incoming chain.

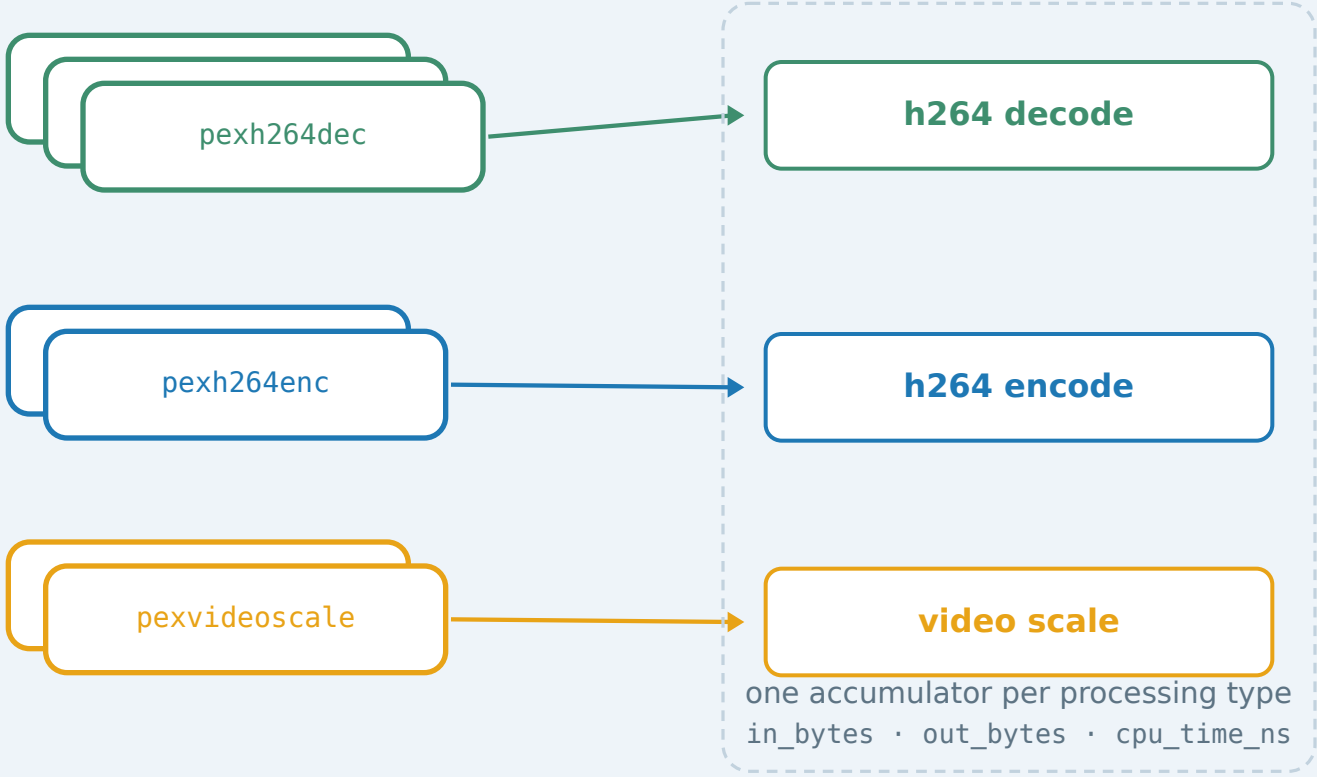
Every thread has its own CPU counter, so a start taken on an input thread cannot be subtracted from an end taken on the worker thread.

Where to keep the results

We have a suitable API and we can measure a thread's CPU time. Now the numbers have to go somewhere.



The counter cannot live in the element itself: elements come and go at any moment, taking the accumulated numbers with them.



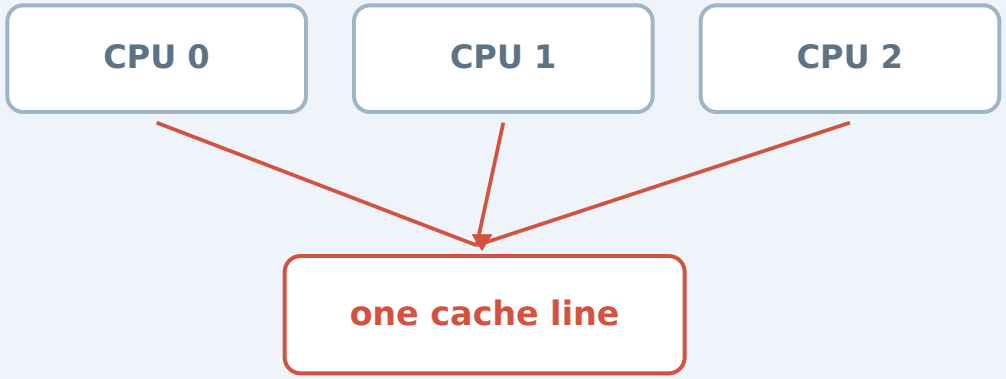
1 **How do many threads write into it at once?**

2 **How does a hook find the right accumulator fast?**

Problem 1: many threads writing to one place

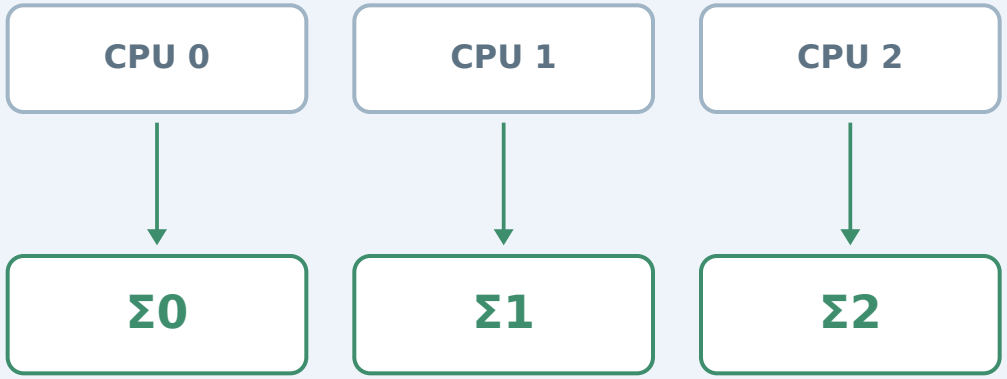
Incrementing a single counter is the ideal case for an atomic. But the profiler showed a drop exactly here.

shared atomic



the atomic solves lost updates, but every writer contends for the same cache line

per-CPU shards



writes are spread over separate cache lines; a rare reader sums the shards

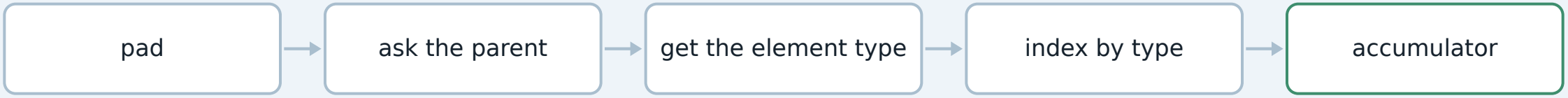
ns per counter update, by number of writing threads

threads	1	2	4	8	16	32
shared atomic	17	46	99	255	542	1375
per-CPU shards	17	18	22	26	29	44

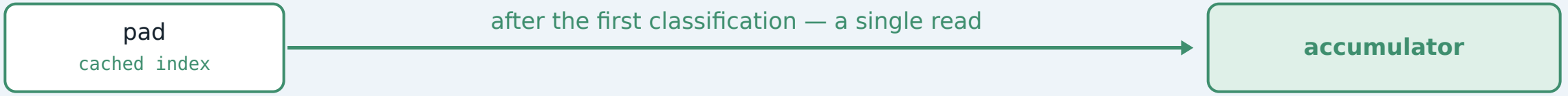
The atomics stay: a thread can migrate between picking a CPU and writing.

Problem 2: finding the right accumulator from a hook

In the hook context all we have is the pad the event happened on.



and that happens on every hook call — that is, on every buffer on every observed pad



Where the cache lives

For now — in a reserved slot of the GstPad struct. That is a tie to our own build, not a recommendation.

Why not GObject qdata

Reading it takes an internal lock — far too expensive on every hot call.

Long path 9–23 ns, reading the cache 2–5 ns — 4–5x cheaper at every thread count.

g_object_get_qdata lands at 13–21 ns — caching there costs as much as doing the lookup again.

Do not read the clock if nobody is listening

While profiling the hooks we found one more spot: GST_TRACER_TS was evaluated before we knew whether anyone had subscribed.

before

```
ts = GST_TRACER_TS;
l  = lookup (key);
nl = lookup (NULL);
call (l, nl, ts);
```

after

```
l  = lookup (key);
nl = lookup (NULL);
if (l || nl) {
  ts = GST_TRACER_TS;
  call (l, nl, ts);
}
```

This is the monotonic event timestamp, not our per-thread CPU counter. For an active callback it is still evaluated.

A point nobody listens to: 20.9 → 3.9 ns, 5.3x cheaper.

The clock read alone is 17.5 ns. On the points we do subscribe to: 21.4 → 21.5 ns, no regression.

The change is small, but it runs at every observation point in the process.

What we would like from the API

Part of it is already done in our build, part is a wish list for the Tracing API.

already done in our build

- compile-time hook categories
- timestamp only when listeners exist

what is still missing

- opt-out for the timestamp
- cheap tracer state on a pad
- direct task hooks

Separately for the pad-probe API: a phase after chain and push return.
In the Tracing API those post points already exist — and they are exactly what we use.

> But no new hook on its own will cover the internal thread pools of third-party libraries: such code still needs its own accessible observation points.