Evolving sched_ext: Resource Control, Topology Awareness, and Energy Efficiency for Modern Systems

Changwoo Min and Gavin Guo
{changwoo,gavinguo}@igalia.com

April 15th, 2026

igalia

Quick Recap on LAVD

- **What is the LAVD scheduler (originally)?**
  - LAVD: Latency-criticality Aware Virtual Deadline scheduler
  - Motivated by gaming workloads
  - Primary target: Windows games running on Linux (SteamOS)
  - Use waker/wakee frequency as a primary hint to decide task's urgency
  - Implemented based on the sched_ext framework (BPF + Rust)
- **There were some improvements for energy model aware scheduling.**
- **Now, we are expanding LAVD as a default fleet scheduler.**
  - Found two major areas for improvement:
    - cpu.max support (cgroup v2)
    - More sophisticated load balancing

Talk Outline

- **Cpu.max support for SCX schedulers**
  - What is cpu bandwidth control (cpu.max)?
  - How is it implemented in the kernel?
  - What are the design goals of SCX cpu.max?
  - How SCX design differs from the kernel implementation?
- **Improve load balancing in LAVD**
  - LAVD is a domain DSQ scheduler
  - Limitations of the current LAVD load balancer
  - LAVD LLC Domain load abalancer
  - Per-domain queue load (util_est tracking)
  - Capacity-proportional fair share mechanism
  - Symmetric budget-controlled migration
  - Future work: Selective migration: size + preemption vulnerability

What is CPU bandwidth control (cpu.max)?

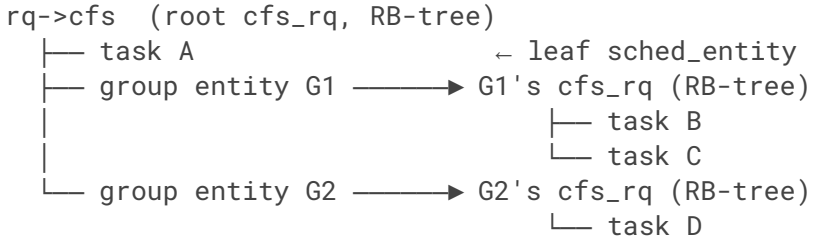
- **Multi-tenant systems (containers, VMs, cloud workloads) need hard CPU quotas.**
  - Over-quota tasks steal CPU from neighbours
  - Under-enforcement wastes purchased capacity.
- **cpu.max = ($QUOTA, $PERIOD, $BURST) in cgroup v2**
  - A cgroup may use at most $QUOTA ns of CPU time in $PERIOD
  - Tasks in an over-quota cgroup are parked until the next period boundary.
  - Underspent quota can be carried over to the next period up to $BURST

The overall cpu.max flow in kernel

- Each cgroup has a quota pool refilled every period.
- CPUs pull slices from the pool on every dispatch.
  - When the pool runs dry → cgroup is throttled.
  - Throttled tasks are dequeued and parked.
- At the next period boundary → cgroup is unthrottled.
  - Tasks are re-enqueued.
- Key design aspects:
  1) How to select a task given cgroup hierarchy?
  2) How to detect if a cgroup goes beyond the quota?
  3) How to replenish a cgroup periodically?

Task selection: nested RB tree

- Cgroup hierarchy is mirrored to the nested red-black tree structure.
- Task selection == walking down the nested red-black tree.



- Cost of task selection: O(log N) ⇒ O(D x log N)
  - where N is # of runnable tasks and D is the cgroup depth.
- Task selection cost increases linearly as the cgroup hierarchy becomes deeper.

Throttle detection: synchronous pull model

- Each CPU manages its local time budget (e.g., 5 msec) borrowed from the global cgroup quota.
- Task execution consumes the local time budget and its ancestors' local budget.
- When the local budget drops to zero, the CPU tries to pull a new slice from the global quota.
- If the cgroup's global quota is also exhausted, the task will be throttled.
  - The cgroup and all its descendants are throttled transitively down the hierarchy.
- The throttle detection cost also linearly increases as the cgroup hierarchy becomes deeper.
  - And, it requires to access the global pools in the critical path.

Replenishing a cgroup: two per-cgroup timers

- Each cgroup maintains two timers: `period_timer` and `slack_timer`.
  - Period timer fires at every $PERIOD and replenishes the cgroup's $QUOTA.
  - Slack timer is to return unused local budget to the global pool asynchronously.
- The number of timers increases linearly as the number of cgroups grow.

SCX cpu.max: design goals

- **Task selection: O(log N), no additional overhead**
  - Task selection is performance critical, so let's avoid any additional overhead.
- **Throttling detection: O(1), not accessing memory hot spot**
  - Throttling detection should be cheap without touching the global memory hot spot for performance and scalability.
- **Replenishing a cgroup: not too many timers**
  - Two timers per cgroup is too many.
  - Can we come up with a simpler design?

SCX cpu.max: key design ideas

- **No nested RB tree for a runqueue**
  - All unthrottled tasks share the runqueue (DSQ in the SCX term).
    - So, there is no change in task selection – still O(log N).
  - Each cgroup manages a runqueue for throttled tasks – namely BTQ (backlogged task queue).

  ⇒ Task selection and throttling is just a regular unnested RB tree operation, so it is O(log N).

- **Eventual bandwidth control**
  - Kernel's synchronous pull model is designed for accurate and immediate control within a period.
    - But it is expensive especially in the hot path (e.g., task selection).
  - Alternatively, let's aim for eventual bandwidth control across multiple periods.
    - If a cgroup overspent a quota, let it pay the debt in the following periods.
    - This opens the door for the off-the-critical-path design.

SCX cpu.max: key design ideas

- **Off-the-critical-path design**
  - Run the heavy-duty jobs (e.g., precise hierarchical accounting) in background.
  - So, we can make the critical path (e.g., task selection) as cheap as possible.
- **Shared timer for all cgroups**
  - Normalize $QUOTA to the standard period (100 msec).
  - A single timer can handle all cgroups.

SCX cpu.max: BPF library

```
Kernel SCX
  │  ops.cgroup_init()            ← BPF callback
  │  ops.cgroup_exit()
  │  ops.enqueue()
  │  ops.dispatch()
  │  ops.tick()
  ▼
BPF scheduler                     → Library: lib/cgroup_bw
  ├── lavd_cgroup_init()          → scx_cgroup_bw_init()
  │     init a cgroup
  ├── lavd_cgroup_exit()          → scx_cgroup_bw_exit()
  │           exit a cgroup
  ├── lavd_cgroup_enqueue()       → scx_cgroup_bw_throttled()
  │     check if a cgroup is throttled or not
  ├── lavd_cgroup_dispatch()→ scx_cgroup_bw_reenqueue()
  │     reenqueue throttled tasks if the cgroup is unthrottled now
  └── lavd_cgroup_tick()          → scx_cgroup_bw_consume()
            report time used for a cgroup
```

Key data structures (simplified)

```
1.  struct scx_cgroup_ctx {
2.      /* --- read-only cacheline --- */
3.      u64  nquota;                /* quota normalised to CBW_REPLENISH_PERIOD (100 msec) */
4.      u64  nquota_ub;             /* min(nquota, ancestors' nquota): effective limit */
5.      bool has_llcx;              /* true if per-LLC contexts are allocated */
6.
7.      /* --- read-write cacheline --- */
8.      bool is_throttled;          /* hot-path throttle flag */
9.      s64  period_budget;         /* effective quota: nquota_ub + burst credit - debt */
10.     s64  runtime_total_sloppy;  /* aggregated LLC runtime (may lag) */
11.     s64  runtime_total_last;    /* runtime_total at last replenishment */
12.     u64  avg_consumption_rate;  /* moving average of CPU time consumption rate */
13. };
14.
15. struct scx_cgroup_llc_ctx {
16.     u64        id;              /* cgroup id */
17.     s64        runtime_total;   /* cumulative CPU time consumed (ns), since replenish */
18.     scx_atq_t *btq;             /* backlog task queue for throttled tasks */
19. };
```

Quota model: one timer for all cgroups

- User configures arbitrary {`$QUOTA, $PERIOD, $BURST`} per cgroup
- We normalise every quota to a fixed 100 msec window:
  - `nquota`: i.e., normalized quota
  - All cgroups now speak the same unit.
  - One shared replenishment timer serves all cgorups simultaneously.
- Manage a cgroup's effective upper bound
  - `nquota_ub = min(nquota, tightest ancestor's nquota)`
  - Hierarchical effective limit (i.e., hierarchical quota in the kernel term)
- Manage the actual budget for this interval (100 msec)
  - `period_budget = nquota_ub + burst_credit - debt`

Runtime accounting: LLC-local atomic

- `scx_cgroup_bw_consume(cgrp_id, consumed_ns)` called from:
  - `ops.tick()`: every ~1–4 ms while the task runs (incremental)
  - `ops.stopping()`: final accounting when the task is descheduled

- One atomic increment on an LLC-local counter
  - `__sync_fetch_and_add(&llc_ctx->runtime_total, consumed_ns)`
  - No cross-LLC cacheline contention

- The accounting timer later aggregates LLC counters into the cgroup total (`runtime_total_sloppy`).

Async two-step throttle: off-the-critical-path

- At hot path, call `scx_cgroup_bw_throttled(cgrp_id)`.
  - `READ_ONCE(cgrp_ctx->is_throttled)`
  - O(1), no lock, no atomics, no hierarchy walk

- A dedicated **accounting timer** fires every 1–20 ms (adaptive).
  - Bottom-up pass: walk the cgroup tree post-order
    - Aggregate LLC runtime_total into each cgroup's `runtime_total_sloppy`
    - If `runtime_total_sloppy >= period_budget`, set `is_throttled=true`.
  - Top-down pass: walk the cgroup tree pre-order
    - For each non-throttled cgroup, walk up the ancestor chain
    - If a cgroup is throttled, throttle all its descendents.

Async but accurate: debt carry-over

- A cgroup may overspend for up to one accounting interval before the flag is set.
  - However, the accuracy is not sacrificed.
- Overspend becomes **debt** at the next replenishment boundary:
  - `debt = max(runtime_consumed - current period_budget, 0)`
  - `next period_budget = nquota_ub + burst_credit - debt`
- The reduced budget in the next interval exactly recovers the overspend.
- Long-run average CPU usage converges to the configured quota.
- We trade detection latency for dispatch efficiency. We do not trade accuracy.

Adaptive accounting timer: keep the detection latency small

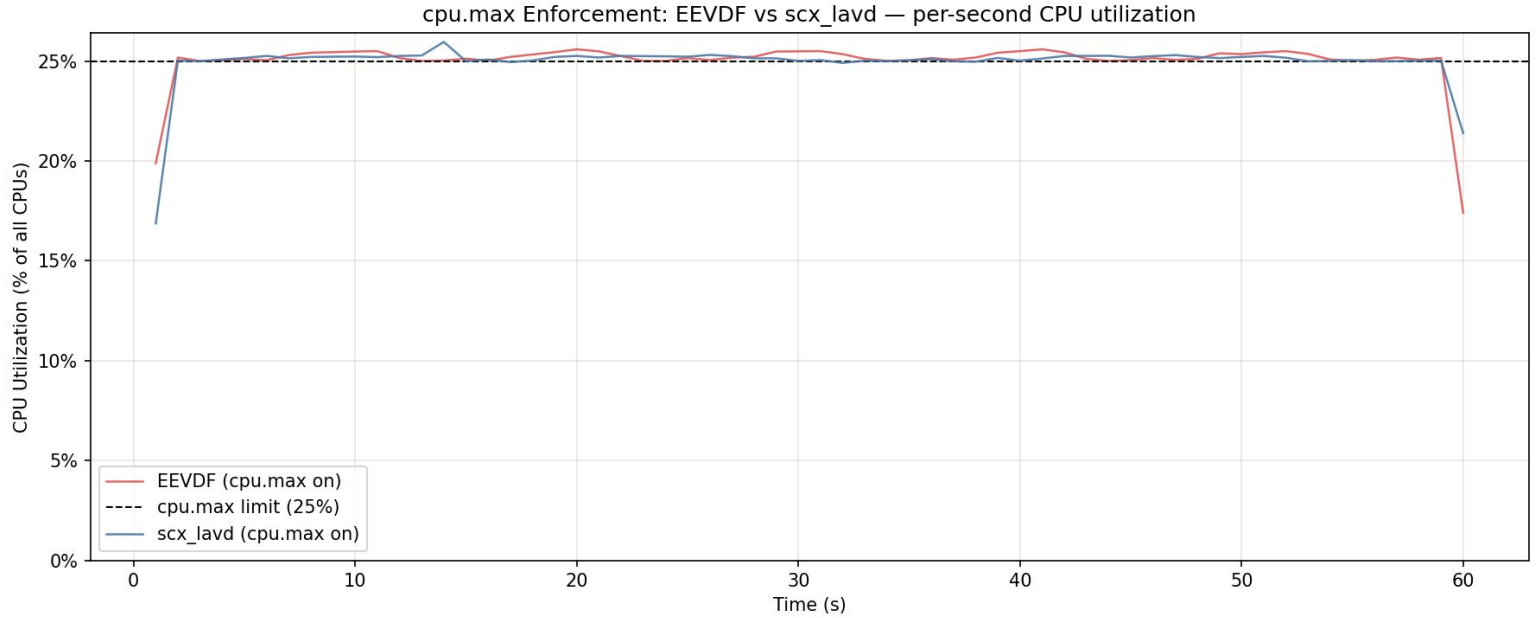
- Each cgroup tracks consumption rate as an EWMA (`avg_consumption_rate`).
- Accounting timer predicts when a cgroup is likely throttled:
  - `remaining_budget = period_budget - runtime_total_sloppy`
  - `time_to_throttle = remaining_budget / avg_consumption_rate`
- Next accounting interval:
  - `clamp(time_to_throttle/4,` 1 ms, 20 ms).
- Timer fires 4 times before predicted exhaustion
  - catches rate changes early, keeps the overspend window small in practice.
- Idle cgroups preserve their historical rate → fast detection on wake-up.

Replenishment: top-half + bottom-half

- **Top half (BPF timer context)**:
  - Every 100 ms, the replenishment timer fires.
  - Compute `period_budget` for each cgroup (debt, burst credit).
    - Update `is_throttled`.
  - Prepare to unthrottle cgroups.
    - Publishes list of cgroups needing BTQ drain.
- **Bottom half (`ops.dispatch()` context)**:
  - Pops tasks from BTQs, re-enqueues to normal DSQs.
- Keeps the timer callback fast; distributes drain work across CPUs naturally.

Results: bandwidth enforcement

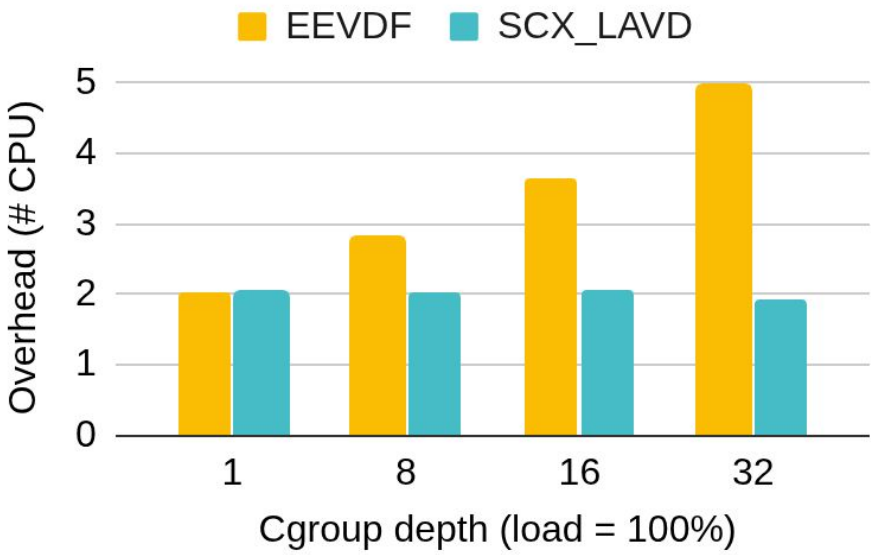
- CPU bandwidth enforcement comparison between SCX vs. EEVDF
  - 2x AMD EPYC 96-core processor (192 CPUs)
  - stress-ng –cpu

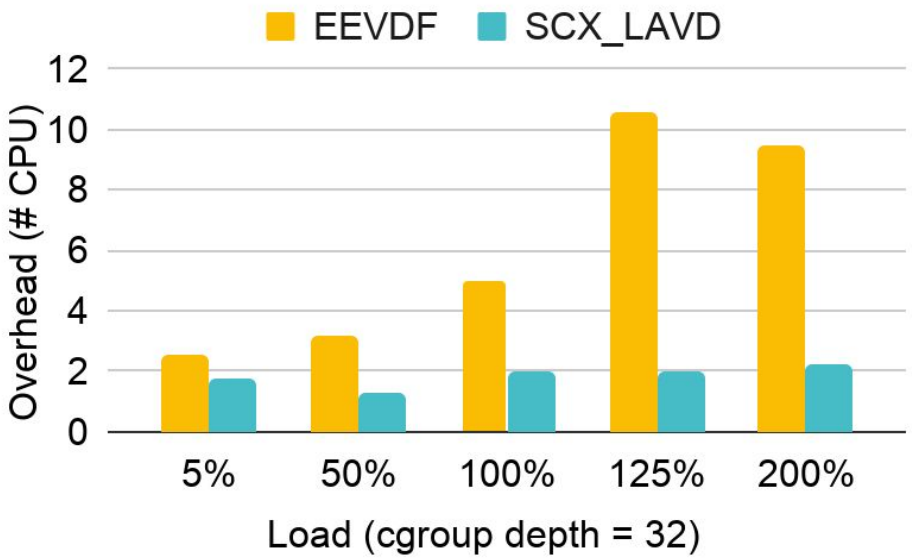


Results: scheduler overhead

- Measurement methodology
  - Run `stress-ng –cpu` on userspace and measure how much time spent on the kernel space to measure the scheduler overhead.
  - Set the cgroup quota to ($nproc)
  - Change the cgroup depth and load
  - 2x AMD EPYC 96-core processor (192 CPUs)

Results: scheduler overhead





Summary

	Kernel	SCX cpu.max library
Throttle check per dispatch	O(D) in `pick_task_fair()`	O(1), flag read
Task selection	O(D x log N)	O(log N)
Throttle detection	Synchronous, on every dispatch	Async, off critical path
Long-run accuracy	Exact	Exact (via debt carry-over)
Timer count	2 per cgroup	2 in total

where D is cgroup depth and N is # of runnable tasks

Talk Outline

- **Cpu.max support for SCX schedulers**
  - What is cpu bandwidth control (cpu.max)?
  - How is it implemented in the kernel?
  - What is the design goals of SCX cpu.max?
  - How SCX design differs from the kernel implementation?
- **Improve load balancing in LAVD**
  - LAVD is a domain DSQ scheduler
  - Limitations of the current LAVD load balancer
  - LAVD LLC Domain load abalancer
  - Per-domain queue load (util_est tracking)
  - Capacity-proportional fair share mechanism
  - Symmetric budget-controlled migration
  - Future work: Selective migration

LAVD == Domain DSQ scheduler

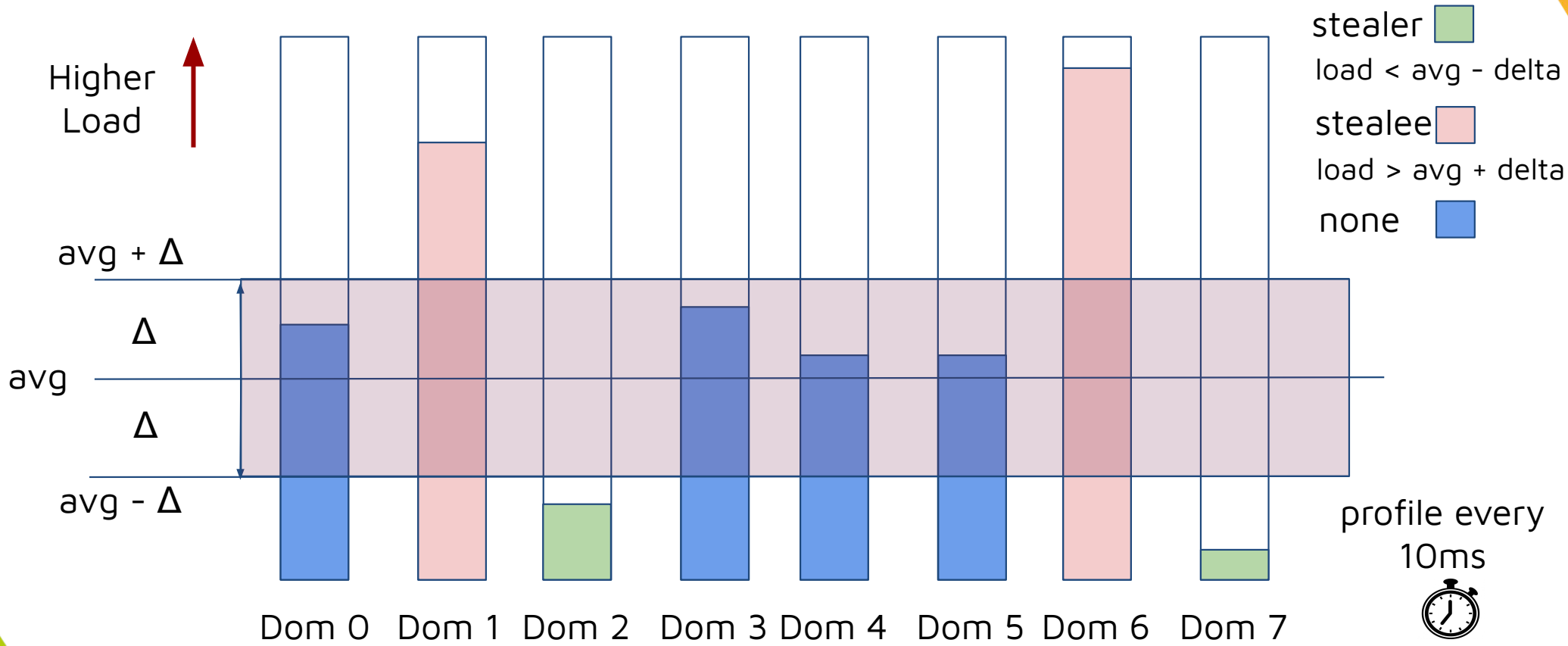
- LAVD is a domain DSQ scheduler where each domain has a dispatch queue shared with the CPU inside the domain.
- A domain takes either stealer or stealee role:
  - ***Stealer*** — underloaded domains who periodically pull tasks from the Stealee domains to keep its CPU busy.
  - ***Stealee*** — overloaded domains having more queue load than its fairshare and stealers can pull task from it.
- When a CPU's local rq drains, ops.dispatch path attempts:

  1. Stealers steal from a remote stealee domain (load balancing)
  2. Try the local domain's DSQ
  3. Force-stealing from any nearby DSQ

Limitations of the Current LAVD Load Balancer

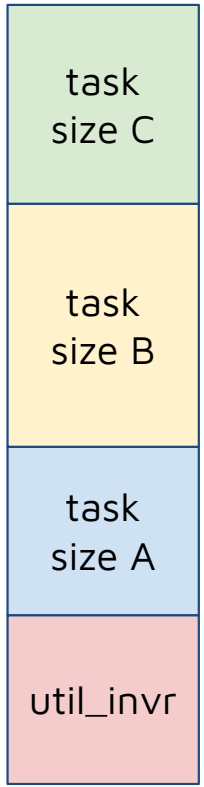
- The load metric is **util + scaled queue length** and not **task size**.
  - A domain with 10 tiny tasks looks busier than one with two long-running tasks.
- No budget control on migration volume.
  - **Thundering-herd** stealing: many CPUs race to drain the same overloaded domain in one round with probabilistic gating.
- No task-type preference during migration.
  - Large task can be scheduled on the small cores.

LAVD Domain Load Balancer



Per-Domain Queue Load

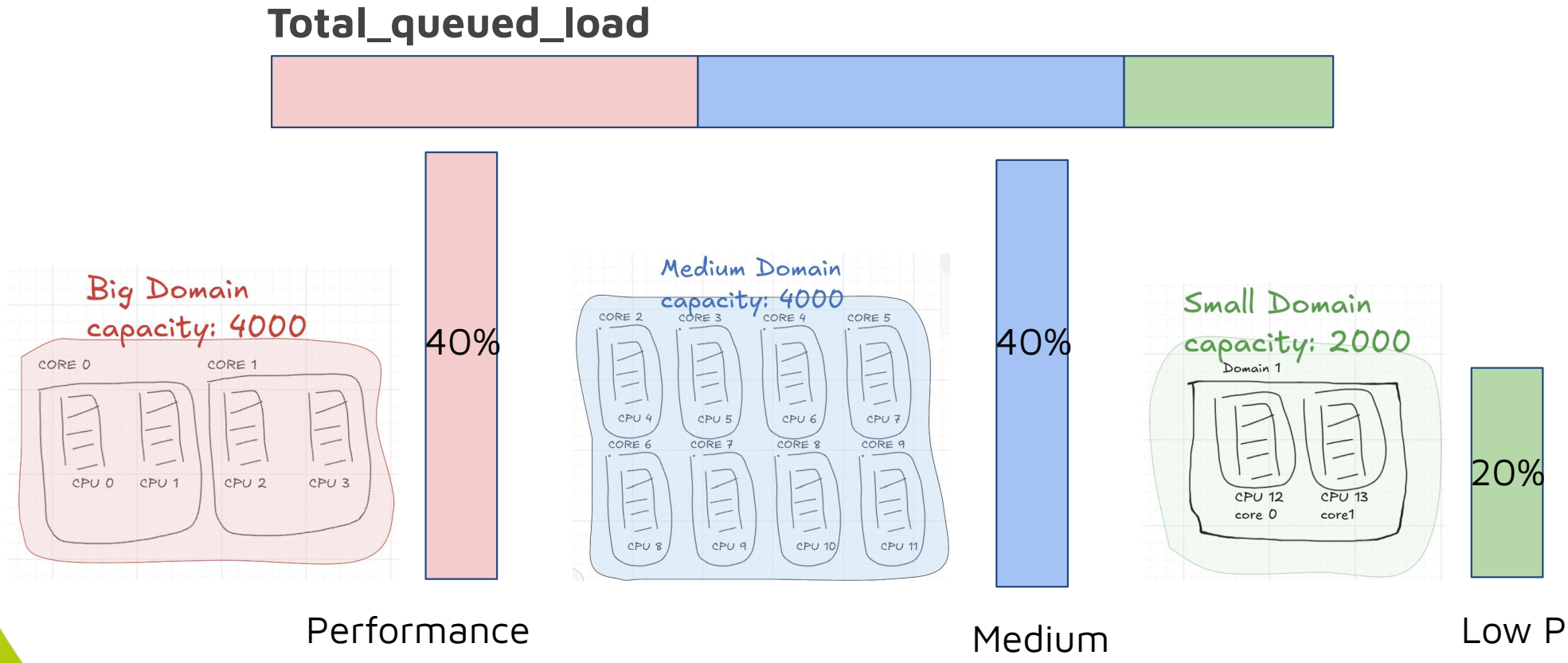
- Unlike task count, this captures actual workload—a domain with two heavy tasks is correctly recognized busier than the one with ten small tasks.
- invr = invariant runtime, scaled by CPU capacity and frequency; so loads are comparable across heterogeneous cores
- **Per-domain load** = **queued_load_invr + util_invr**
- **queued_load_invr** = **Σ size_of (task)** # for all tasks queued in the domain
- **Util_invr**: how busy the domain's CPUs currently are



Fair Share Mechanism

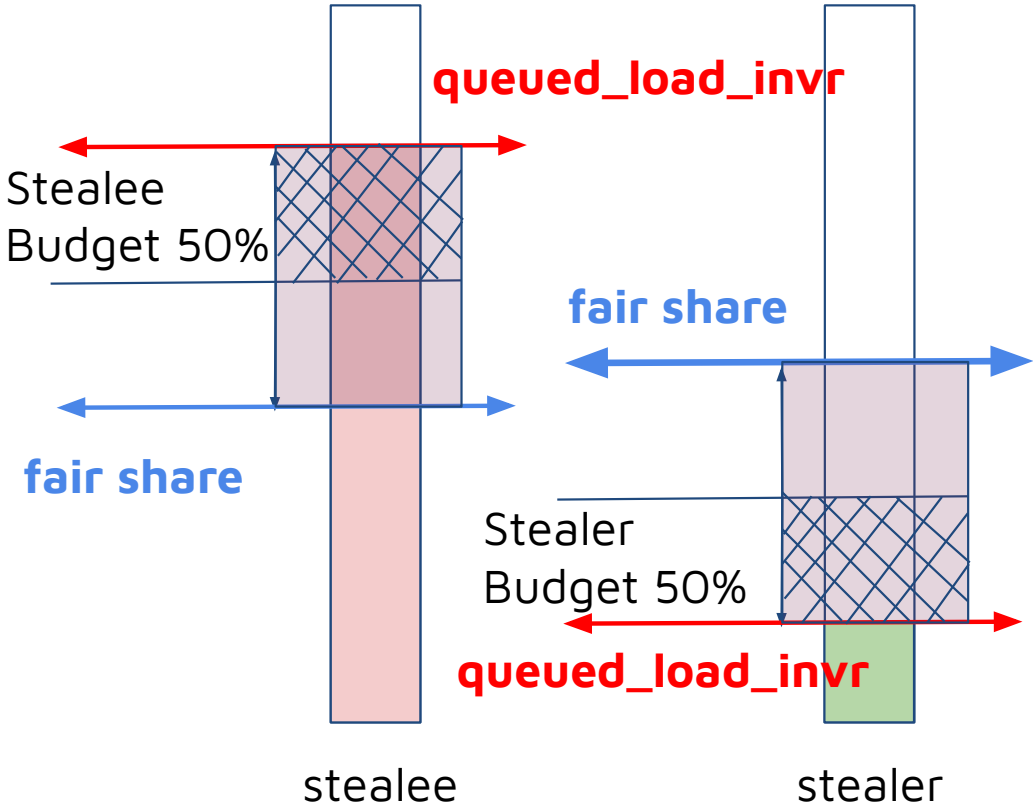
- Fair share: how much queued load a domain should carry, proportional to its capacity
- **fair_share = total_queued_load_invr × (domain_capacity / total_capacity)**
- For example, a domain with 40% of the system's capacity should carry 40% of the total queued load. On heterogeneous systems, big-core domains have higher compute capacity and therefore a larger fair share.

Fair Share Mechanism



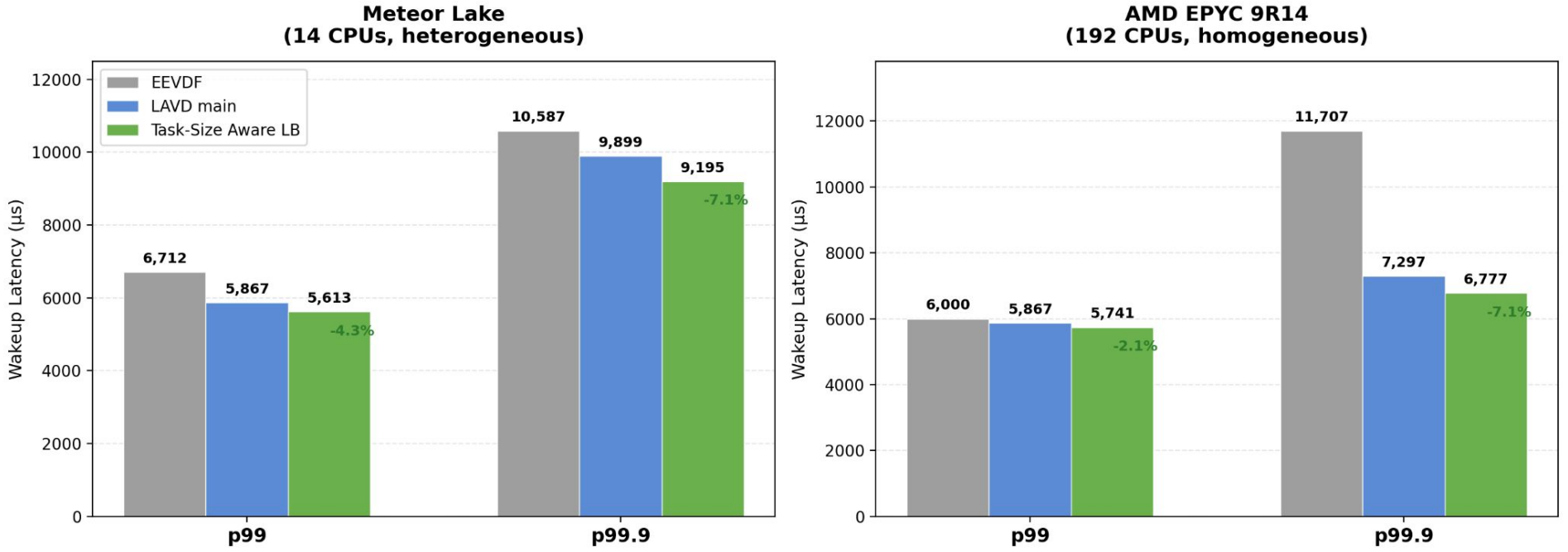
Budget-Controlled Mechanism

- ***Migration budget***: half the excess load beyond fair share—prevent overshooting. Moving everything in one round may trigger the imbalance and ping-pong effect.
- ***Stealee***: An overloaded domain allows half of its excess load to be migrated.
- ***Stealer***: An underloaded domain allows half of its deficit to be received.



Evaluation Results

schbench Wakeup Latency: EEVDF vs LAVD main vs Task-Size Aware LB



schbench -m 4 -t 8 -r 30 (Meteor Lake) | schbench -m 8 -t 24 -r 30 (EPYC) | 6 runs each | Green % = Task-Size Aware LB vs main

Future Work

- Even with the task-size aware migration, current stealing always picks from the head-of-DSQ — it may **not** be the best candidate to migrate.
- So, let's migrate the most appropriate task => selective migration.
- Preferred to be migrated:
  - To choose the best tasks, Peek the processes from the count 2–8?
- Preferred **not** to be migrated:
  - Task-Hot Guard — skip tasks still within the cache-hot threshold (analogous to CFS's task_hot() at ~500 μs)

Future Work: Selective Migration

- What factors should be considered in the Selective Migration?
  - Task size — match task to domain capacity?
  - Latency criticality?
  - Cache locality grouping?
  - Waiting time? A long-waiting process may need to be rescued to avoid long tail latency?


igalia