Optimizing Google Search and beyond with pluggable scheduling

LINUX PLUMBERS CONFERENCE
Vienna, Austria
Sept. 18-20, 2024

Barret Rhoden
Josh Don

01

A brief history of pluggable scheduling @ Google

Motivation

Kernel Rollout Speed

Updating a large fleet of machines is a slow process, can take O(months) to update all machines due to disruption SLOs.

↓

Updates via userspace are fast!

Kernel Programming Constraints

Would be nice to avoid kernel constructs like per-cpu runqueues.

↓

In userspace/BPF we can more easily implement any kind of policy we want.

Lack of Specialization

Number of sched classes is small, and each one has to be fairly generic.

↓

With pluggable scheduling, we can easily create application specific policies.

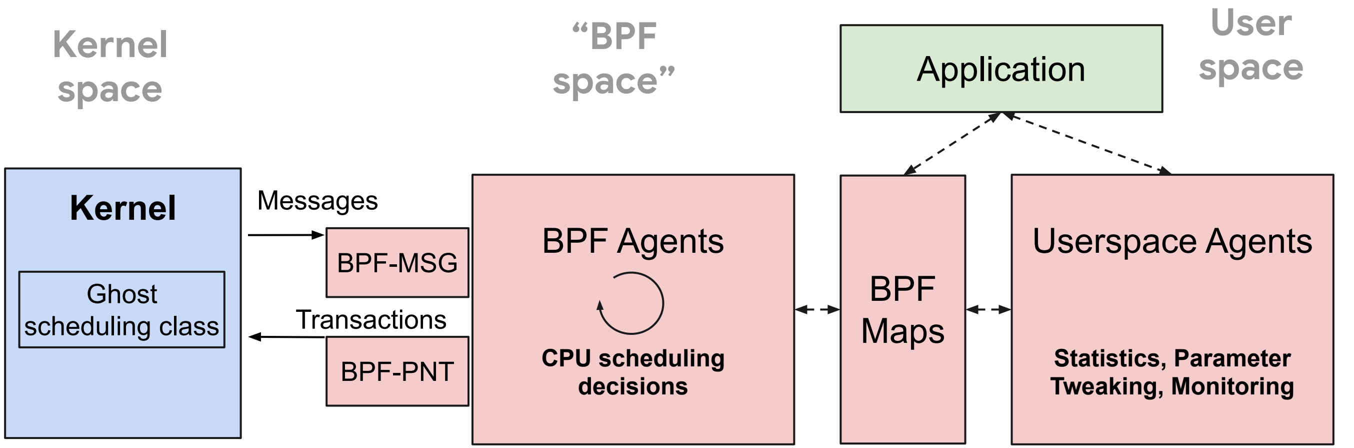
Milestones and future direction

- **2019** — ghOSt development kicks off
- **2020** — Initial experiments with internal workloads
- **2021** — SOSP paper published
- **2022** — Architecture shift heavily towards BPF; more advanced policy development.
- **2023** — sched_ext collaboration with Meta
- **2024** — ghOSt deployed for specific workloads such as Search. **Starting to move policies to sched_ext**.

02

Building a scheduling policy for Google Search

eBPF Scheduling Model



from "eBPF Kernel Scheduling with Ghost", LPC '22

Search from 1000 ft.

- Numerous **groups** of threads:
  - Query: handle RPCs, various types. O(100us - 10ms)
  - Pollers: important, **but noisy**. O(10us)
  - Housekeeping: various jobs
- Sensitive to **cache locality** and CCX placement
- **Latency** sensitive
- Run on multi-socket (NUMA) machines

from "ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling", SOSP '21

Policy: cpu soft partitioning

- Numerous cpus, several thread groups...
  - Let's **spatially partition** the machine among those groups
- Two benefits:
  - **Cache locality** for each group (RPCs, Pollers)
  - Isolate the Pollers to minimize their O(us) **interference**
- Not using cpu masks (affinity is too "stiff")
  - Each cpu is assigned to a group: if you want it, you get it ("Dibs")
  - Can use cpus assigned to other groups if they don't want it
  - Readjust the assignment periodically from userspace

Policy: per-CCX runqueues

- Each cpu is assigned to a CCX runqueue, many-to-one
- High L3 **cache locality**, but less cpu / core locality
- Rapid assignment of threads to cpus
  - No head-of-line blocking
  - No waiting for the load balancer
  - Cuts down on **latency**
- Still try to keep threads on their previous cpu, but don’t wait

Policy: app-specific pick_next_task()

- Different policy for different types of thread (e.g. RPC, Poller)
- RPCs:
  - Each CCX's runqueue sorts threads by their **RPC deadline** (EDF)
  - Search tells us the deadline via a giant BPF ARRAY_MAP (soon, an arena!)
  - Keep tasks on cpu until complete. **No quanta** or time slicing!
- Pollers:
  - Just get on any cpu quickly. (FIFO)
  - Future work: throttle their cpu partition if the app says they waste cycles.

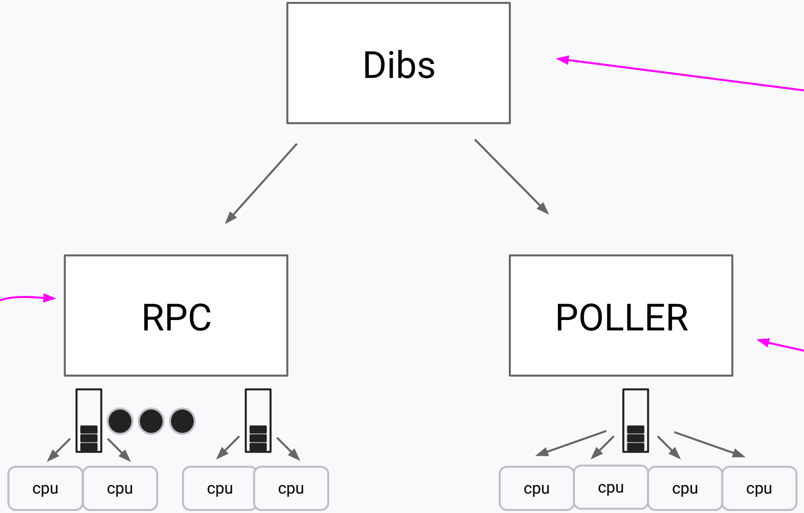
The array map and other stuff from "eBPF Shenanigans with Flux", LPC '23

Putting it all together: Flux

- Expressed this Search policy as a hierarchy of schedulers
  - Numerous cpus, several thread groups
  - Each thread group gets its own scheduler (struct + code)
- Flux is a framework for writing hierarchical schedulers and for composing multiple scheduling policies
- Top of hierarchy: scheduler of *subschedulers*
  - Soft partitioning of **cpus** to **groups of threads**
- Leaf: typical thread scheduler
  - Assigns **threads** to **cpus**: Group / Application-specific policy

Flux is from "eBPF Shenanigans with Flux", LPC '23, and the Ghost repo

Putting it all together: Flux



Assign cpus to subschedulers (groups of threads):

- Soft partition cpus
- nr_cpus based on child's load
- Pack into CCXs for locality

Schedule threads onto cpus:

- Per CCX runqueues
- EDF Policy

Schedule threads onto cpus:

- Single runqueue
- FIFO policy

Results

- Single node benchmark, testing data, compared to CFS:
  - 10% more QPS
  - 27% less p50 latency
  - 14% less p90 latency
  - 3% less p99 latency

03

General purpose scheduling policies and CFS

What about general purpose scheduling?

- We cannot write a bespoke policy for everyone (and most don’t need one)

What about general purpose scheduling?

- We cannot write a bespoke policy for everyone (and most don't need one)

1. Can we write a general purpose scheduler that improves on tradeoffs made in CFS?
2. Can we layer policy on a machine to support some tasks with a bespoke policy, and everything else on a general purpose policy?
3. Can any of the lessons we learn be applied to CFS?

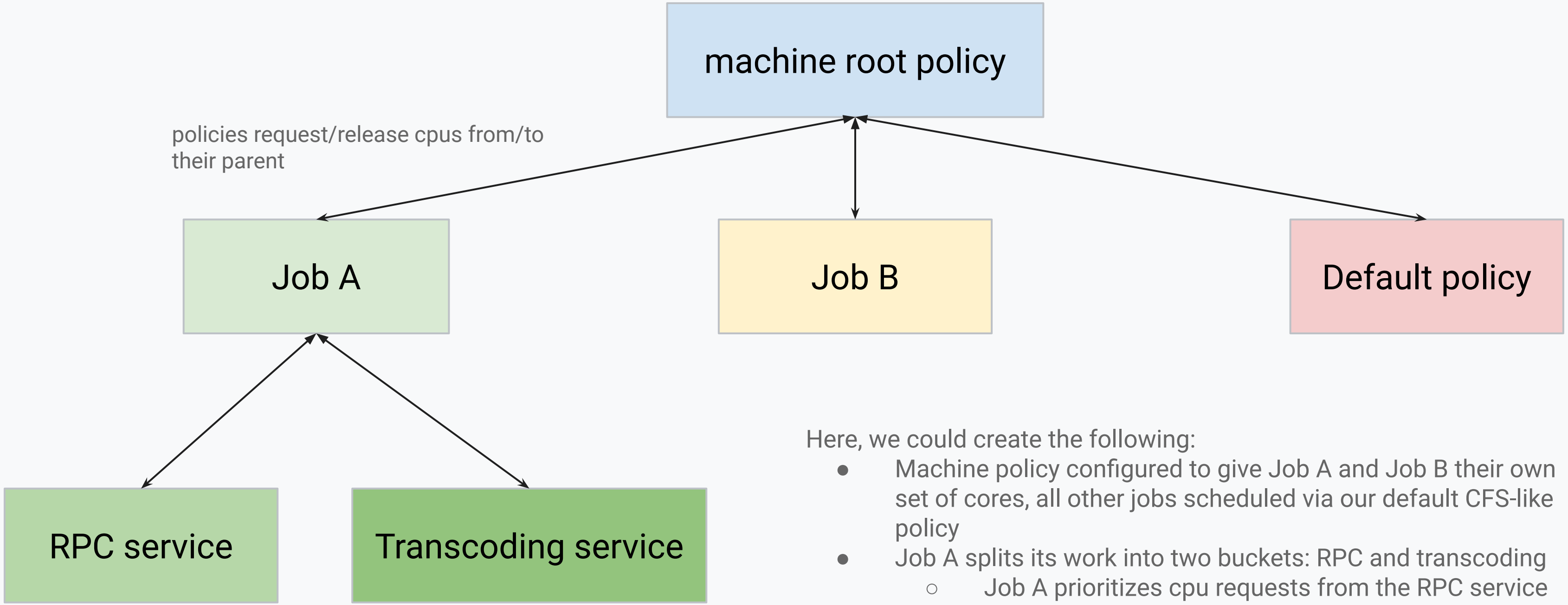
How to Improve upon CFS

- Treat cgroups as a first-class entity => map groups of tasks to groups of cpus
  - Threads of a job stay more closely together, rather than spray to all cpus
  - Soft affinity (prefer cpus X-Y, but allow spillover)
  - Fits nicely with chiplet architecture

- Iterate on policy to help enforce latency bounds
  - Experiment with deadline-driven mechanics, such as EEVDF

- Tuning scheduling knobs
  - Replicating existing knobs (migration_cost, wakeup_latency, etc.) and tuning with ML
  - Experimenting with new knobs

How do we Layer Policy



Here, we could create the following:

- Machine policy configured to give Job A and Job B their own set of cores, all other jobs scheduled via our default CFS-like policy
- Job A splits its work into two buckets: RPC and transcoding
    - Job A prioritizes cpu requests from the RPC service
    - RPC service focuses on fast, FIFO handling of inbound RPCs
    - Transcoding service focuses on batch, cpu bound, round-robin processing

How do we Layer Policy

- Flux: a scheduler of schedulers, in BPF
- BYOP: Bring your own policy
- Flux allocates cpus to policies
  - Can change the allocation dynamically to share time for multiple policies
  - Policies use callbacks to register a cpu being given or taken away

04
Q&A