select_idle_sibling and wake_affine pains

Xi Wang

Scheduler MC

Linux Plumbers Conference 2020

Overview

- This talk is about
  - Our workloads and performance problems
  - Thoughts, not solutions
  - Warming up for the next talk
    - Joint session with “The Thing” that was “Latency Nice”

select_idle_sibling being the soft^2 rt sched class

- Many applications are latency sensitive yet not specialized enough to fit rt or SCHED_DEADLINE
  - CFS being the only practical choice

- Wake up load balancing is keeping these applications happy
  - When num of running threads <= num of cpus
    - One thread on each cpu -> negligible sched latency -> :)
    - Two threads on the same cpu -> 10ms level (round robin) sched latency -> :(
  - Periodic or new idle load balancing is not frequent enough
  - Search more and spread more usually work for our workloads but opposite problems exist
    - Idle cpu search can add 10s of us to the wake up path, migration cost
      - Lots of cache line bouncing

- select_idle_sibling got the job it never wanted
  - The select_task_rq_fair call tree is like half of the class

Not an ideal solution

- Even a small change reshuffles application performance metrics
  - Affects how close we can track upstream
    - Regressions get more attentions than improvements during rebase
    - Sometimes we have to revert to the old behavior even if the new behavior makes sense
- Only one decision can be made for the wake up path. Optimizations easily step each other

	Spread to Idle	Sched Overhead	Sleep States	Turbo / DVFS	big.LITTLE / asym
Latency	x	x	x		
Throughput	x	x		x	x
Energy Efficiency	x		x	x	x

Problems encountered

- Order of search
  - Search from cpu 0 -> search from target (waker) => worse cache locality
    - Two threads doing ping pong wakeups can walk through all cpus
    - recently_used_cpu was added but further improvements might be possible
- Aggressiveness of wake_affine
  - With WA_IDLE sched feat we had too many threads wake affined to the same irq cpu for a particular application
  - Should hard irq or softirq cpu be considered idle?
    - Timer interrupts can be a separate class because they move with threads (scheduled by threads and fire on the same cpu)
  - We tried to enable wake affine across numa >=2 times but always had problems

Problems encountered

- LLC being the max range of search
    - Too narrow for AMD CCX (4~8 core LLC/NUMA domains)
        - As a cpu socket can have up to 64 cores, inter CCX dynamic imbalance can be a major problem
    - Too wide for some of the latency_nice usage cases due to overheads
- Early search termination (SIS_PROP)
    - One value heuristic, works well with efficient load balancing
    - Less accurate if threads have cpu affinity restrictions

How can we improve?

- Keep improving it in small steps?
  - Relatively straightforward to hack for individual cases, but difficult generalize / upstream
- Customized tuning to the rescue?
  - Evolved “latency nice”
- Explicitly support them?
  - I think the population of server side latency sensitive applications is growing
  - Explicitly support them with an new / existing sched class?
    - If priority or deadline can be enforced, spreading out is less critical
    - Why didn’t they fit an rt class in the first place?
      - Still expect stuff in a generic sched class: Scaling up to many threads, oversubscriptions, work conserving and best effort support etc.