Optimizing low-latency Scheduling

pjt@google.com
Preamble: Cores as a resource container

This is not the focus of the placement work which follows. But a useful model to think about for encapsulating and representing scheduling policy that we think the work described in this talk can benefit.

(The results discussed in the C-state scheduling section did not use or depend on these semantics.)
Preamble: Cores as a resource container

Schedulers traditionally map \textit{threads} to \textit{cpus}.

Recently, we’ve seen \textbf{group scheduling} introduced in mainstream schedulers.

But around a lot of the interesting work we can do here, there’s now a mismatch:

- Many macro policies are difficult to implement against a pointwise thread representation.
- The group representations don’t help us here either, really it’s the equivalent problem.
- New hardware interfaces such as memory bandwidth constraints are being exposed as per-core interfaces.
Preamble: Cores as a resource container

Hyper-threading introduces additional performance variance.

.. and now, additional strong isolation requirements around security and data-isolation -- something that networking can be quite sensitive to.

Direction here:

Cores as a first class scheduling abstraction. Core-aware schedulers and policies. Virtual core APIs.
Scheduling C-states
x86 C-states in a nutshell

- **C0**: CPU is active, i.e. non-idle
  - The hardware also supports P-states for controlling the power drawn by an active CPU. They are disjoint from C-states.

- **C1/E**: A very shallow sleep, think `mwait`. Caches remain powered. Exit latency is small. Core voltage can be dropped in C1E. \( O(1-10\text{us}) \) exit.

- **C3**: The first “deep “state, TLB/L1/L2 caches are flushed. Voltage drop [CRET]. \( O(10\text{us} \text{ of us}) \) exit.

- **C6**: Power-gated (voltage->0). Processor state externally stored and restored on wake. \( O(\sim100\text{us}) \) exit.
Why do we use C-states?

- **Power**
  - Quite simply, we do not have the capacity to run every core in a data-center at Vmax.

- **Turbo**
  - Allows a core to run at greater than nominal frequencies by borrowing its neighbors’ power-budget.

- **Parts lifetime**
  - Power gating means that there are fewer (or none) electrons running through an idle cpu.
Background: Hardware C-states Management

- Prior behavior: hardware C-state auto-demotion
- OS always requests deepest sleep state (C6)
- HW decided whether to use a shallower sleep state

Example flow of HW auto-demotion:

- C0 (Active) -> Idle, OS requests C6, HW picks C1
- C1, C6

Wakeup penalty:

- 230μs
- 100μs
Idea: Consider CPU C-state in wake-up selection

Before:

After:
Pollsters. Getting it right since 2016.

C0

C3/C6 boundary.

Next predicted event.

CPU0

[Shallow C-state, e.g. C1E]

[Deeper C-states, C3+]

Decision made here
Pollsters. Getting it right since 2016.

Next predicted event.

C3/C6 boundary.

Decision made here

CPU2  CPU3  CPU5

X

time
…. And how the pollsters get it wrong

After:

1 2 3 4 5

a

...but 500us later

After:

1 2 3 4 5

a
Idea

- When there are no active CPUs, our choice is still important.
  - We cannot improve the current wake-up. But we can improve future wake-ups by additionally considering our prediction boundaries as a second classifier.
  - Additionally, make cores that we are not directing wake-ups towards, sleep more quickly.

After:

...but 500us later
Packet(s) arrive

RX cpu
HardIRQ

SoftIRQ
E.g. RFS/RPS

Waiting
Thread
e.g. recv(2), epoll(2)

Process receives

Network Processing
"V0"

Trend towards asymptote of all involved cores in deep C-state.

**Legend:**
- RX core leaves C1
Now doing a better job of extending also for RX core
Low latency SSD Service
CPU consolidation visualized
Further Ideas

- Optimizing evaluation cost
- Dynamically manage HardIRQ assignment
- Migratable predictor state, thread “pointwise” representation
- ML around better predictor models