Idle Cycle Injection
in Linux

Salman Qazi
Google Inc.

Static Power Provisioning

In a typical data center, power is statically provisioned.

- There is some finite supply of power available.
- At every level of power distribution, there is a local limit enforced by circuit breakers.
- Racks and clusters populated to respect these limits in the worst case scenario
- On average, far less power is used than allowed by the infrastructure.

Dynamic Power Provisioning

- Goal: Get useful work out of as much power as possible
- Put more racks into clusters and machines into racks than permitted by worst case scenario
- Dynamically determine power quota, while staying within the limits
- Enforce the quota on individual machines by power capping

Power Capping

- Limiting the amount of power used by an individual machine
- Many different approaches in hardware, firmware and software
  - e.g. DVFS can be used to set an upper bound on CPU power usage.
  - "Power Capping a prelude to Power Shifting", Lefurgy et al., does it in firmware.
- Has other applications, such as responding to thermal emergencies.
- Indiscriminate and agnostic of workloads, when implemented without software integration.

Modelling Power

- The maximum power used by the CPU increases, as CPU usage increases
- The maximum power used by RAM increases, as CPU usage increases
- We model the power used by disks as a constant.
- It is sufficient to control CPU usage in order to limit peak power.
- However, additional information about RAM and disks will help make the bounds tighter.
- See: "Power provisioning for a warehouse-sized computer", Fan et al.

Idle Cycle Injection

- Run an idling instruction on the CPU X% of the time, over every interval of length Y.
- "Power capping via forced idleness", Gandhi et al.
- Advantages:
  - Simple and widely available
  - Allows the use of C-states
  - Fine granularity of control - compare to statically picking P-states
  - Software solution: Flexibility to discriminate between tasks
- Disadvantages
  - Software solution: Prone to OS bugs

Idle Cycle Injection - contd.

Our Design Choices

- Avoid contention
  - Simplest way - Control each CPU independently
- Whenever possible, account for natural idleness
- Avoid affecting latency of interactive tasks
- Provide flexibility to the user to control which tasks run when power is limited.

Accounting for natural idleness

- Naive algorithm - On each CPU:

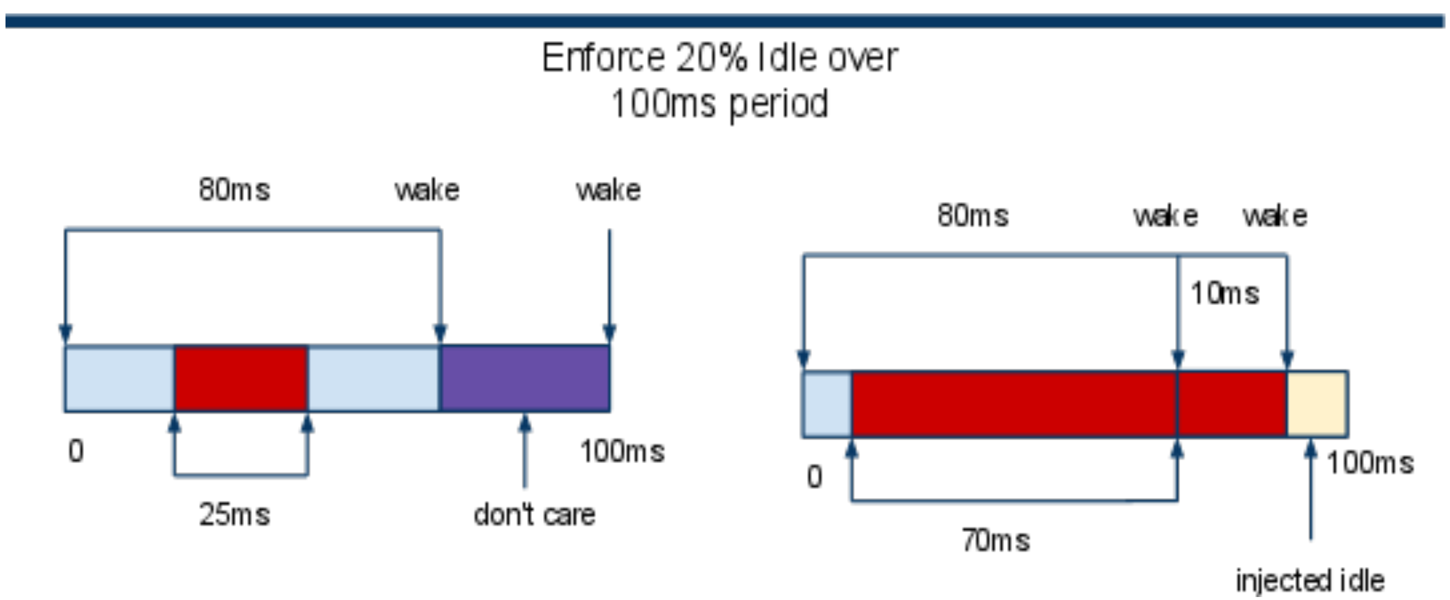
```
while (1) {
        schedule_for(interval * (100 - min_idle_percent) / 100);
        inject(interval * (min_idle_percent) / 100);
}
```

- Unnecessarily prevents useful work.
- Instead:

```
while (1) {
    time_left = monitor_cpu();
    inject(time_left);
}
```

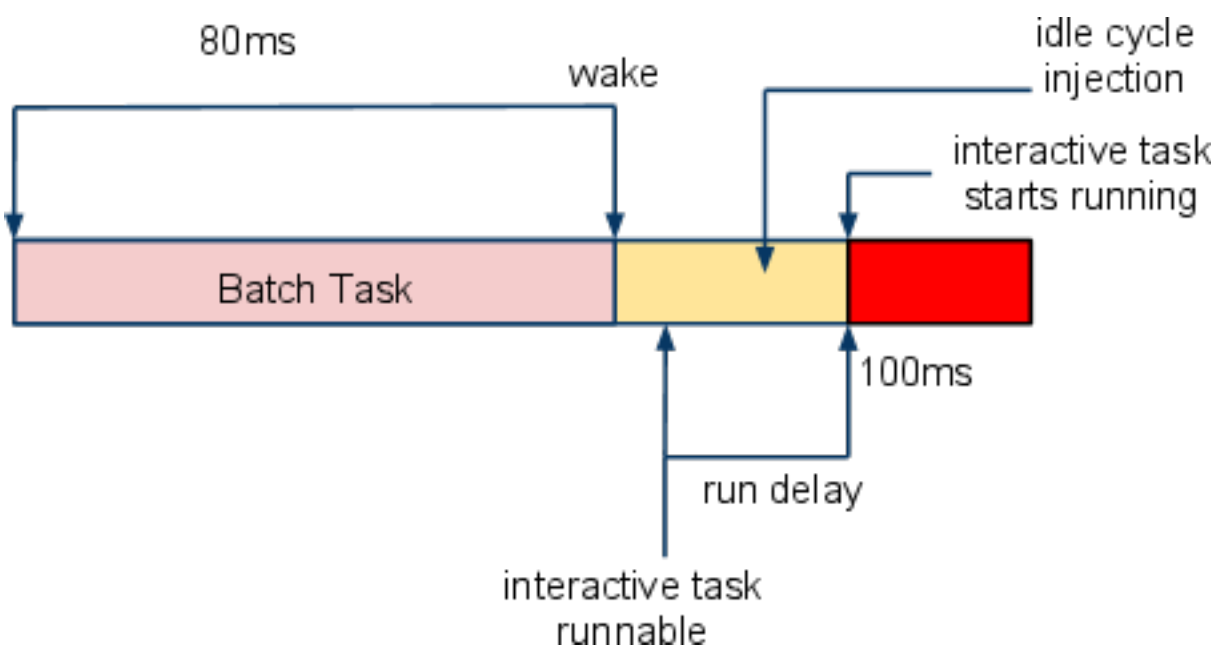
monitor_cpu

- Let Z be the minimum of the remaining interval and the remaining CPU time for the interval.
- If Z is smaller than timer granularity, then
  - Return the remaining interval.
- Otherwise,
  - schedule work until now + Z and repeat.



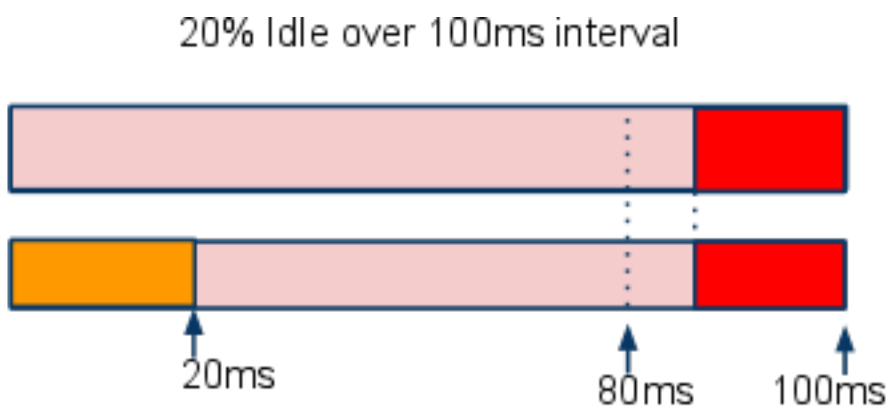
Problem for interactive tasks

If a batch task uses up the entire CPU quota, the interactive task cannot run until the next interval.



Solution

Eagerly inject up to the minimum number of idle cycles when there are no interactive tasks on the run queue.



Algorithm

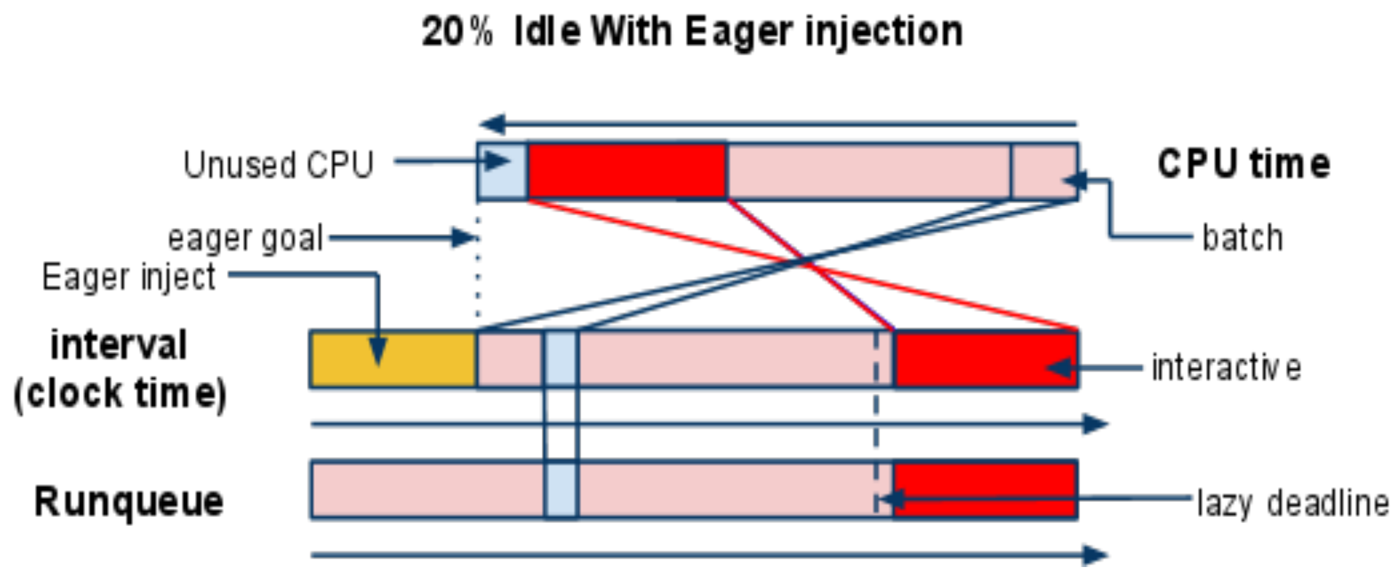
```
/* interval_left: remaining interval in "clock time" */
/* cpu_left: remaining interval in "CPU time" */

monitor_cpu:
        if (min(interval_left, cpu_left) < timer_granularity)
              return interval_left
        Z = get_next_timer()
        T = set_timer(Z)
        while (timer_pending(T)) {
           schedule_while((mode != EAGER ||
                  interactive_on_runqueue)
                 && timer_pending(T))
           eager_inject();
         }
         goto monitor_cpu

eager_inject:
       while (mode == EAGER && !interactive_on_runqueue &&
            timer_pending(T))
                 do_idle();
```

Algorithm cntd...

```
get_next_timer:
    lazy_deadline = min(cpu_left, interval_left)
    eager_goal = interval_left - cpu_left
    if (eager_goal > 0) {
        mode = EAGER
        next_timer = min(lazy_deadline, eager_goal)
    } else {
        mode = LAZY
        next_timer = lazy_deadline
    }
    return next_timer
```



Discriminating Between Tasks

- virtual runtime (vruntime) is the time consumed by a task scaled down by its relative weight.
- CFS picks the task with the lowest vruntime
- Since Idle Cycle Injector is a real time thread, CFS does not know about its time consumption.
- Blame CFS tasks for that time.
    - Choose which tasks to blame according to a user specified order.
    - A task can only be responsible for at most its fair share during injection period.
    - Reshuffle the runqueue.

Laptops in Deserts

- These techniques can be applied to extend battery life when power is unavailable
- Predictable power savings
- Ability to select "important" applications
- Accounting for the power constraint in scheduling decisions
- Potential for simple but effective interfaces

Possible Extensions

- P-state interpolation
- Machine-wide cap
- Take advantage of newer hardware
    - Better models: don't just guess power

Acknowledgements

The Idle Cycle Injector has been a joint effort between myself and Ken Chen.

Lots of good ideas and other support from other people at Google, both in the kernel team and outside.

Fin

Questions?