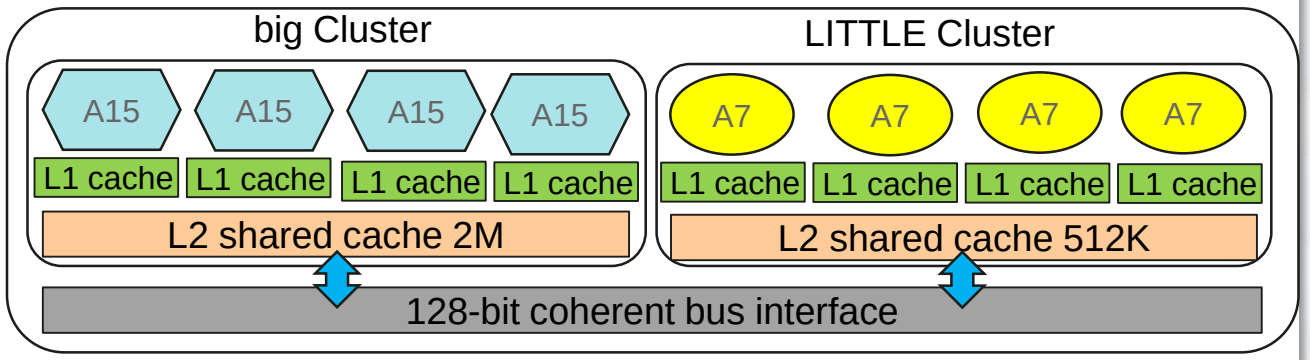
ITMO Advanced OS 2024

Aleksei Romanovskii, PhD,
SPb Research Center (CBG OS Lab)

Lesson 2024.12.04

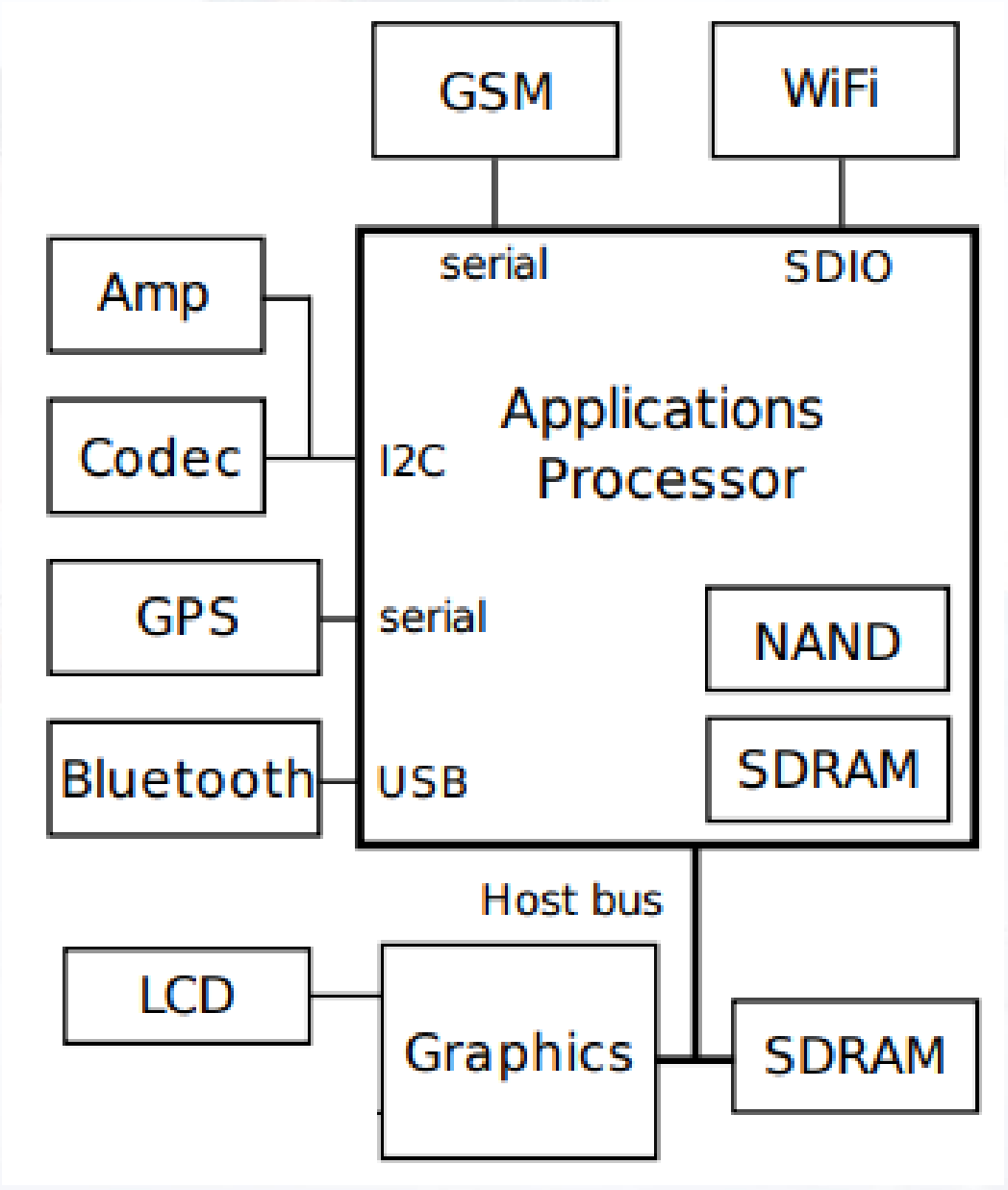
Device abstraction 1 of 2

- Clustered Heterogeneous Multicore Architecture (HMA)
  - O is set of |O| cores, indexed as o[1..|O|] : O = {o[1],…o[|O|]}
  - H is set of |H| core types, indexed as h[1..|H|]
  - K is set of |K| clusters, each comprises cores of the same core-type, indexed as k[1..|K|]
- Each core-type cluster operates on independent frequency, while cores in the same cluster operates in the same frequency
  - Frequency range for core-type is defined by a finite set F of |F| frequencies, indexed as f[1..|F|]
- Each cluster has an independent Last Level Cache (LLC) that is shared by all cores in the cluster
  - LLC can be partitioned into set A of |A| cache partitions, indexed as a[1..|A|]
  - Core based partitioning: each core in a cluster can be assigned a number of distinct a[i]
  - Alternative task based partitioning: a number of distinct a[i] can be assigned to each task (see Task model)
  - Set W of |W| cache partitions assigned to each core (task), indexed as w[1..|W|]
- SoC example in terms of the specified model:
  - |O| = 8 cores,
  - |H| = 2 (ARM big.LITTLE),
  - |K| = 2
  - Fbig = {0.2, 0.3, …2.0}GHz, Flittle = {0.2, …, 1.4}GHz
  - LLCbig = 2048kB, LLClittle = 512kB. |A| = 8



Device abstraction 2 of 2

- GSM – cellular radio
- Amp - audio amplifier
- I2C – Inter-Integrated Circuit Protocol/Bus
- SDIO Secure Digital IO
- Serial – IO bus
- Etc.



Task abstraction, 1 of 2

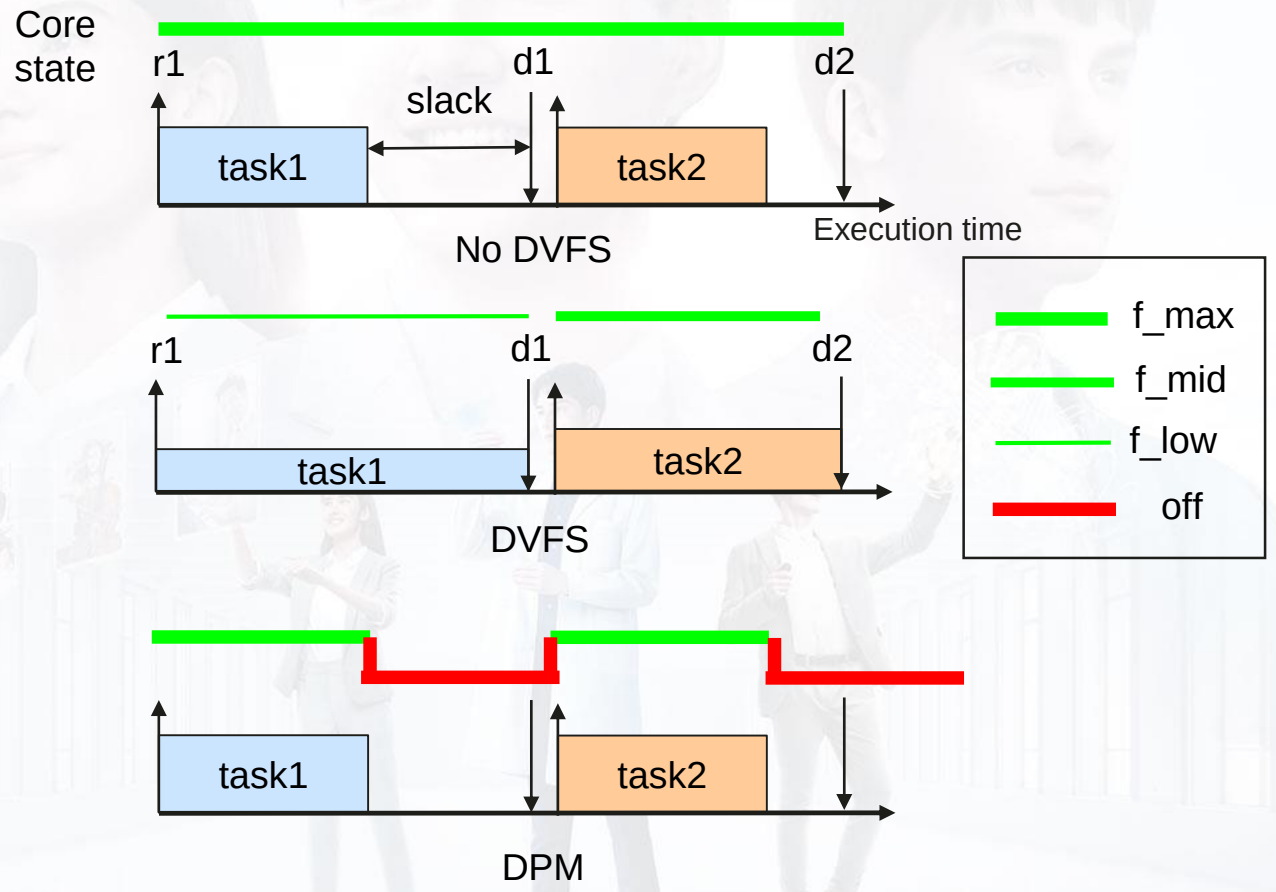
- Open model of non-periodic online tasks
  - new tasks can be added at any time, old tasks deleted at any time
  - each task can be run at arbitrary time
  - each task may have requirements for responsiveness on external events (e.g. on UI)
  - Set T of |T| tasks, indexed as t[1..|T|]: T = {t[1],…t[|T|]}
- Each task t[i] is represented by <r[i], e[i], d[i], p[i], uc[i], um[i], ud[i], un[i]>, where
  - r[i] is release (earliest start) time of task t[i]
  - e[i] is Worst Case Execution Time (WCET) of task t[i]
  - d[i] is deadline (latest end) time of task t[i]
  - p[i] is task t[i] priority (may be dynamic)
  - uc[i] is task average CPU utilization (usually vector for HMA)
  - um[i] is task average memory bandwidth utilization
  - ud[i] is task average disk utilization
  - un[i] is task average network utilization
- Cores (core clusters) are scheduled/partitioned to tasks (or vice-versa). Caches are scheduled/partitioned to tasks
- RAM, disks(IO), network are scheduled/partitioned to tasks

Task abstraction, 2 of 2

- Task t[i] utilization uc[i] = e[i]/(d[i] – r[i]), uc[i] <= 1.
  - To ensure valid schedule the sum of all uc[i] allocated to each core (for a given period of time) must be <= 1
- Task t[i] execution cycles c[i] = cc[i] + mc_cache[i] + mc_ram[i] + dc[i] + nc[i], where
  - cc[i] are compute (CPU core) cycles, used by the task t[i]
  - mc_cache[i] are data access delays, measured in compute cycles, when data is in LLC
  - mc_ram[i] are data access delays, measured in compute cycles, when data is in RAM (either to read in LLC, or to write-through)
  - dc[i] are disk delays for data IO, measured in compute cycles
  - nc[i] are network delays for data IO, measured in compute cycles
- Task execution time e[i] expressed via CPU core frequency f[j], cache frequency f_cache and RAM frequency f_ram: e[i] = cc[i]/f[j] + mc_cache[i]/f_cache + mc_ram[i]/f_ram
- Task metrics that may be used by scheduling algorithm – to have higher IPC
  - compute intensity metric cim[i] = cc[i]/(cc_max[i] + mc_cache_max[i] + mc_ram_max[i] + dc_max[i] + nc_max[i]), where
    - cc_max[i] are compute cycles at max core frequency
    - mc_cache_max[i] are memory cycles at max cache frequency and max |W| of cache partitions allocated to task
    - mc_ram_max[i] max RAM frequency
    - etc.
  - …other metrics that will facilitate task allocation strategy to determine the best core, core-type, RAM amount, IO time slots, network time slots for each task – ultimately to have higher IPC

Power management abstractions, 1 of 2

- Dynamic Voltage and Frequency Scaling (DVFS)
  - clock-gating technique that varies the voltage and frequency of processor cores between minimum and maximum bounds
  - the processor core frequency has direct relation to voltage. Decrease in voltage (and frequency) result in reduction of DPC.
  - Frequency and voltage change at discrete steps. Usually done by programmable voltage regulator and clock generator
  - Inter-task DVFS and intra-task DVFS
- Dynamic Power Management (DPM) to manage power in core idle state - not in our area of interest



Power management abstractions, 2 of 2

- Dynamic Voltage and Frequency Scaling (DVFS), continued
  - Task execution time depends both on core cycle and on memory cycle latency, so it may happen if DVFS increases CPU core clock-cycle length then less memory cycles would suffice, therefore higher IPC could be achieved, see Fig.1
  - In turn Fig.2 illustrates dependency of task Dynamic Power Consumption on DVFS: lower frequency results in lower DPC

Fig.1

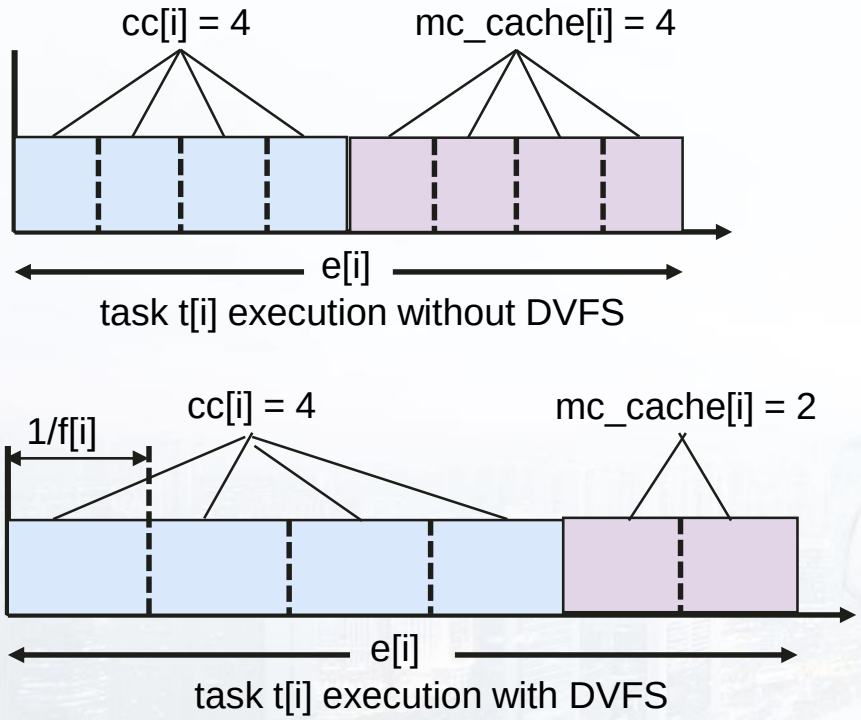
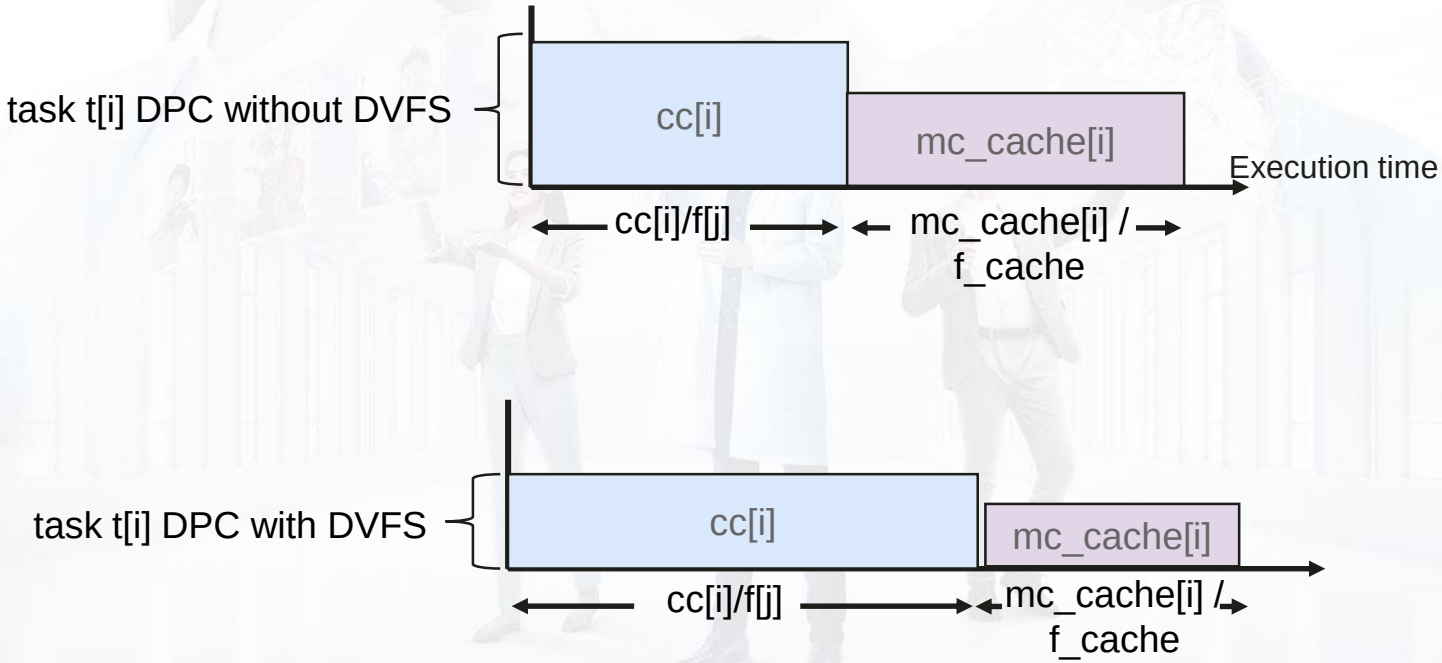


Fig.2



OS CPU scheduler and power formulas

- Mobile devices derive the energy required to operate from batteries that are limited by the size of the device.
- The ability to manage energy usage requires a good understanding of where and how the energy is being used.
- The advancing functionality of modern smartphones is increasing the pressure on battery lifetime, and increases the need for effective energy management.
- Formulas
  - Power: **P(watts) = I(amps)*V(volts)**
  - Energy: **E(joules) = P(watts) * T (sec)**: e.g. kWattHour
  - Battery: **WattHours = AmpHours * Volts**: e.g. 5000 mah * 3.7mV
  - CPU/RAM: **P = f(hertz)*C(farad)*V^2 + Ps**,
    - where **Ps = $P_{short\text{-}circuit}$ + $P_{leakage}$** is static power dissipation
    - **f** and **Ps** are functions of V and Vth (the threshold voltage). In practice voltage, threshold voltage, and frequency are always changed together.

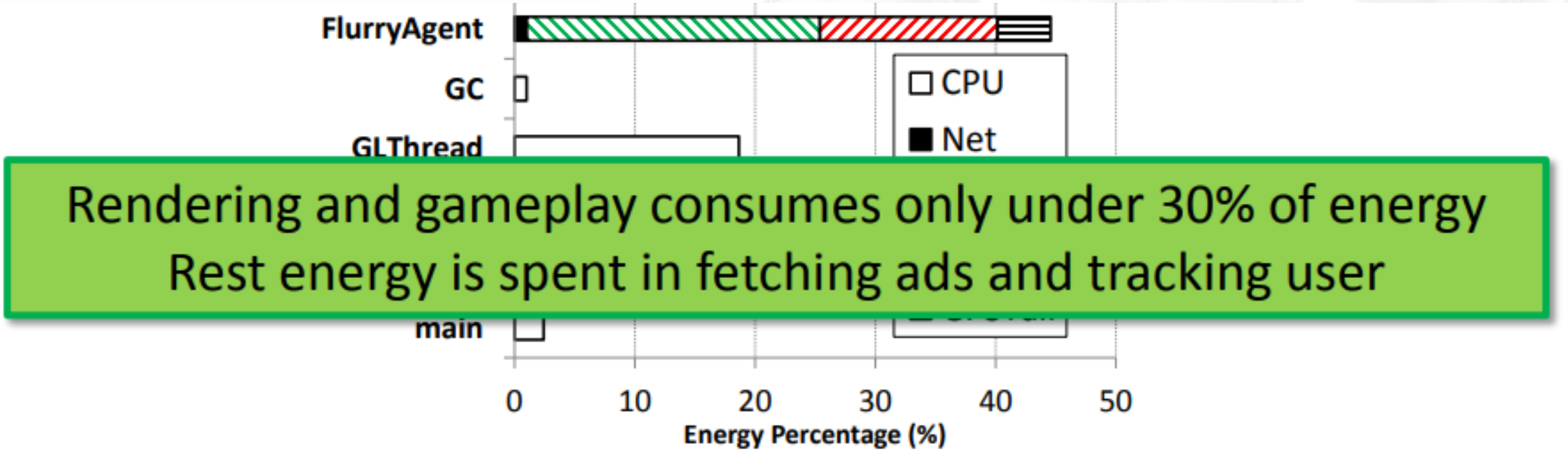
CPU (RAM) power drain

- The core of a CPU contains a clock, which will run at a particular multiple of some frequency generated by a highly stable reference oscillator.
- A phase-locked loop (PLL) is used to generate one or more faster or slower clocks from this reference clock.
- By throttling back the clock speed, the switching rate for transistors is slower, which also reduces the power consumed by the CPU.
- If the core voltage is not changed, then the power consumed per clock cycle is unchanged, but the total power consumed over a given time interval will be lower: **P = f(hertz)*C(farad)*V^2 + Ps**, **E(joules) = P(watts) * T (sec)**

Mobile device power drains (ref.numbers)

- Screen on: 200ma (+touch controller), 300ma (max brightness)
- WiFi on: 2ma, 30ma(active), 100ma(scan)
- Audio DSP: 10ma
- Video DSP: 50ma
- Camera: 600ma, flash +200ma
- GPS: 10-20ma, +50 (signal), +100-300ma (sx,rx), + 1.5ma (scanning, paging)
- Mobile data: 100-300ma
- CPU (core): 3ma (idle/suspend), 50ma (awake), 100-200 (active)
- RAM: 3-25ma (standby), 80-100 (read-write, refresh)
- Bluetooth ?

Case study: Angry birds energy consumption



Rendering and gameplay consumes only under 30% of energy
Rest energy is spent in fetching ads and tracking user

Dynamic Voltage and Frequency Scaling

- Dynamic voltage and frequency scaling (DVFS) is the adjustment of power and speed settings on a computing device's various processors, controller chips and peripheral devices to optimize resource allotment for tasks and maximize power saving when those resources are not needed.

- DVFS allows devices to perform needed tasks with the minimum amount of required power. The technology is used in almost all modern computer hardware to maximize power savings, battery life and longevity of devices while still maintaining ready compute performance availability.

- An unused smartphone, for example, should revert to a low-power mode, barring interference from applications and spyware.

- Multimedia requires more power, so the device reaches a higher power state and creates more heat during heavier processing such as video and gaming.

- Were it not for DVFS, many devices that are passively cooled would require active cooling. However, the noise, bulk and power consumption required by active cooling makes it impractical for smaller devices.

- DVFS helps maintain operable parameters with increased mobility.

big.LITTLE in a nutshell for task scheduling

- The key is task placement
  - Wrong task-core distribution kills big.LITTLE advantages
- big.LITTLE puts high requirements on scheduler
  - It should be aware of 2-x types of cores
  - It should be energy aware
  - it should communicate with the DVFS subsystem
- big.LITTLE scheduling implies heuristics
  - The task placement decision should ideally be made basing on the task's future activity

Scheduler for big.LITTLE

- CFS is a good scheduler
  - But it's not really a perfect fit for big.LITTLE
- Extend CFS to be applicable to non-SMP architectures
  - Work started back in 2013
- 2 competing implementations were developed
  - Qualcomm/Codeaurora (HMP scheduler, QHMP)
  - Linaro/ARM (EAS)

History of Linux CPU schedulers

- Early days: very simple scheduler based on a circular queue (Round-Robin)
- Linux 2.4: O(n) scheduler
  - divide processor time into epochs.
  - Within each epoch, every task can execute up to its time slice.
  - If a task does not use all of its time slice, then the scheduler adds half of the remaining time slice to allow it to execute longer in the next epoch
- Linux 2.6.0 to Linux 2.6.22: O(1) scheduler
  - Each process is given a fixed time quantum, after which it is preempted and moved to the expired array.
  - Once all the tasks from the active array have exhausted their time quantum and have been moved to the expired array, an array switch takes place.
  - Because the arrays are accessed only via pointer, switching them is as fast as swapping two pointers. This switch makes the active array the new empty expired array, while the expired array becomes the active array
- Linux 2.6.23(2007): Completely Fair Scheduler
  - BFS (2009) as alternative to CFS
- Linux/Android (2013+): Discussion of EAS started
- Android (2017): Kernel 4.4-4.9 Energy Aware scheduler
- Linux 5.x (2022): Energy Aware Scheduler
  - Due to heterogeneous CPUs

Completely Fair Scheduler

- The main idea is to maintain balance (fairness) in providing processor time to tasks
- CFS maintains the amount of time provided to a given task to determine if it needs balancing
  - the smaller amount of time a task has been permitted access to the processor — the higher its need for the processor is
- CFS maintains a time-ordered red-black tree
  - Instead of run queues as did predecessors
  - Guarantees O(log(N))

CFS operation principles

- Sorts tasks in ascending order by CPU bandwidth received
  - This is where red-black tree comes into play
- The leftmost task off the rb-tree is picked up next
  - It has the least spent execution time
  - So that task gets the CPU to restore balance (fairness)
- Considers all CPUs to be the same
  - Works very well in SMP systems
  - Does not work efficiently in more complicated cases

EAS basic principles

- Task scheduling that considers energy implications
- Decision should be made basing on:
  - System topology
  - E. g. SMP or HMP
  - Power management features
  - CPU Idle states, DVFS
  - Workload for each core
- Work load calculation is basically independent
  - Separate module providing results to EAS

EAS explained 1 of 3

- EAS goal is to minimize energy, while still getting the job done. That is, we want to maximize:
  - performance (ipc) / power(Watt)

- EAS relies on an Energy Model (EM) of the CPUs to select an energy efficient CPU for each task, with a minimal impact on throughput
  - EM is an interface between drivers knowing the power consumed by devices at various performance levels, and the kernel subsystems willing to use that information to make energy-aware decisions.
  - EM considers CPUs only, i.e. no peripherals, GPU or memory. Model data includes power consumption at each P-state and C-state.

EAS explained 2 of 3

- EAS changes the way CFS tasks are assigned to CPUs.
  - When it is time for the scheduler to decide where a task should run (after wake-up), the EM is used to break the tie between several good CPU candidates and pick the one that is predicted to yield the best energy consumption without harming the system's throughput.
  - The predictions made by EAS rely on specific elements of knowledge about the platform's topology, which include the 'capacity' of CPUs, and their respective energy costs.
  - The 'capacity' of a CPU represents the amount of work it can absorb when running at its highest frequency compared to the most capable CPU of the system.
- EAS categorizes processes into four cgroups, being top-app, system-background, foreground, and background.
- Tasks due to be processed are placed into one of these categories, and then the category is given CPU power and the work is delegated over different CPU cores.
  - The top-app is the highest priority of completion, followed by foreground, background, and then system-background.
  - The background technically has the same priority as system-background, but system-background usually also has access to more little cores.

EAS explained 3 of 3

- When waking the device, EAS will choose the core in the shallowest idle state, minimizing the energy needed to wake the device.
- This helps to reduce the required power in using the device, as it will not wake up the large cluster if it doesn't need to.
- Load tracking is also an extremely crucial part of EAS, this information is used to decide frequencies and how to delegate tasks across the CPU, and there are two options:
  - "Per-Entity Load Tracking" (PELT)
  - "Window-Assisted Load Tracking" (WALT)
- WALT is more bursty, with high peaks in CPU frequency while PELT tries to remain more consistent.
- The load tracker doesn't actually affect the CPU frequency, it just tells the system what the CPU usage is at the moment.
- A higher CPU usage requires a higher frequency and so a consistent trait of PELT is that it causes the CPU frequency to ramp up or down slowly.

Load Tracking: what and why

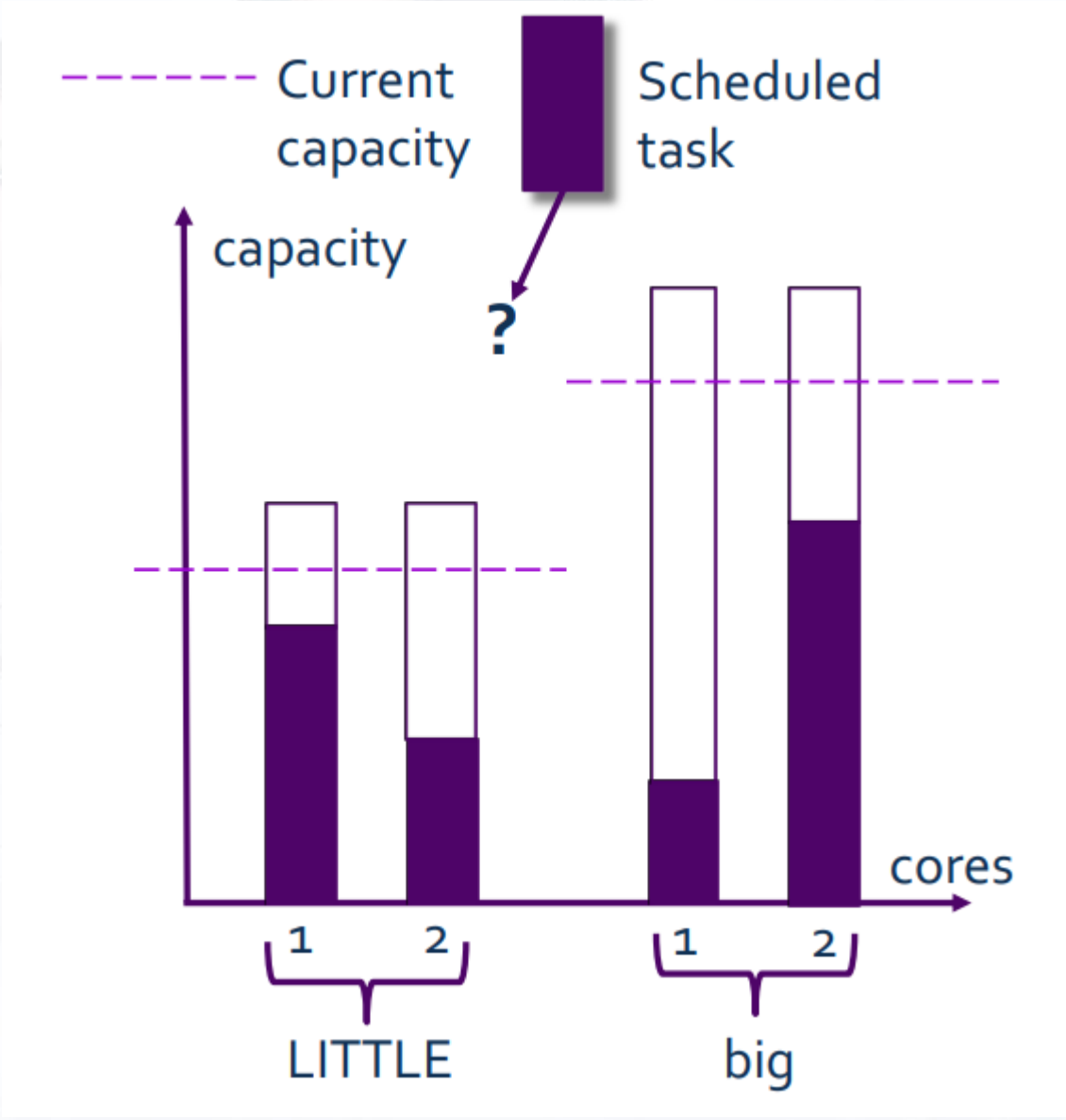
- LT is to track task demand (load) and CPU utilization
  - Classifying a task as "heavy" in mobile world use-cases such as scrolling a UI or browsing a web page where tasks exhibit sporadically heavy load
  - Reclassification of a light task as heavy is also important - for example, a rendering thread may change its demand depending on what is being shown on-screen
- Load balancing
- Task placement
  - Task load and CPU utilization tracking is essential for energy aware scheduling.
  - Heavy tasks can be placed on higher capacity CPUs. Small tasks can be packed on a busy CPU without waking up an idle CPU.
- Frequency guidance
  - CPU frequency governor (schedutil, powersave, ondemand, etc) controls how the CPU raises and lowers its frequency according to the demand.
  - CPU frequency governors like on-demand use a timer and compute the CPU busy time by subtracting the CPU idle time from the timer period.
  - Task migrations are not accounted. If a task execution time is split across two CPUs, governor fails to ramp up the frequency

PELT: Per Entity Load Tracking

- In mainline since kernel 3.8
  - used by mainline CFS
- The main idea is that process can contribute to load even if it is not actually running at the moment
- PELT tracks load on a per-entity (process or task) basis
- Let Li designate the entity's load contribution in period pi
  - Then the total load is L = L0 + L1*q + L2*q^2 + L3*q^3 + ⋯, where q is the decay factor
  - The load is accounted using a decayed geometric series with runnable time in 1 msec period as coefficients.
  - The decay constant is chosen such that the contribution in the 32 msec past is weighted half as strongly as the current contribution.
  - The blocked tasks also contribute to the run-queue load/utilization. The task load is decayed during sleep.

EAS/PELT operation

- Estimate energy
  - E = Pidle * Tidle + Pbusy + Tbusy
- Pick CPU with sufficient spare capacity and smallest energy impact
- Here both LITTLE and big cores have sufficient capacity
- the energy impact is smaller with LITTLE core



Qualcomm HMP scheduler

- Tasks are divided into groups
  - By importance
  - Depending on nice priority
  - By “size”
  - Depending on the calculated load
  - Task may be “big”, ”little” or other
  - Thresholds are parametrized

- Scheduling a task should depend on its properties
  - Task “size” should be defined somehow
  - It’s done basing on task demand calculation

“big” and “small” tasks in HMP

- Small task
  - A periodic task with short execution time
  - Can be easily identified using task average demand

- Big task
  - Task producing high CPU load (parametrized, 90%+)
  - Some heavy tasks HMP doesn’t want to count as big e.g. background threads in Android

- Some tasks are neither big nor small
- Tasks can change their “size” over time

HMP scheduler: task demand

- Task demand Dtask is the contribution of a task's running time to a window
  - Dtask = delta_time*cur_freq /max_possible_freq

  - delta_time - time of task running on a core in a period of time
  - cur_freq - the current frequency of the core this task is running on
  - max_possible_freq is the maximum possible frequency across all cores

- Calculated over N sliding windows (N is a parameter)
  - E. g. the average demand Davg = (D1 + ... + Dn)/N
  - The best result is achieved with D = max{Davg, D1}
  - == Task demand is the maximum of its contribution to the most recently completed window and its average demand over the past N windows.

HMP scheduler: task demand scaling

- We already account for difference in maximum frequency
  - Dtask is calculated in regard to maximum frequency across all cores
- We also need to account for higher performance of big cores
  - Dtask,scaled = Dtask * rq->efficiency / max_possible_efficiency
  - Efficiency is a per-run-queue parameter
  - Usually big cores are considered 2x more effective

EAS/WALT 1 of 2

- WALT: Window Assisted Load Tracking. It retains PELT “per-entity” tracking pattern. It implements N-window demand calculation from QHMP
- WALT keeps track of recent N windows of execution for every task.
- Windows where a task had no activity are ignored and not recorded.
  - Windows exist only when the task is on the run-queue or running. This allows rapid reclassification of the same task as heavy after a short sleep.
  - Thus a task does not need to re-execute to gain its demand once more and can be migrated up to a big CPU immediately.
- Task demand is derived from these N samples. Different policies like max(), avg(), max(recent, avg) are available.

EAS/WALT 2 of 2

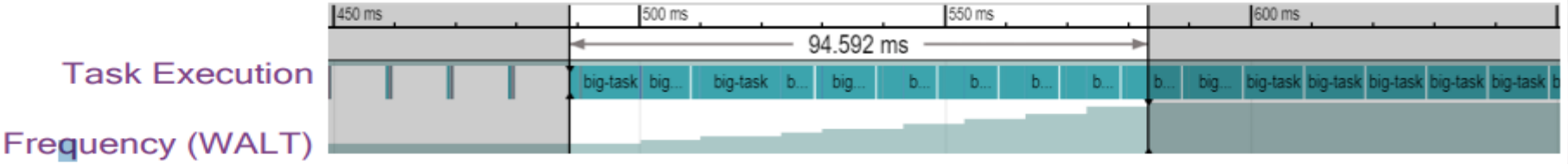
- The wait time of a task on the CPU rq is also accounted towards its demand.
- Task utilization is tracked separately from the demand. The utilization of a task is the execution time in the recently completed window.
- The CPU rq’s utilization is the sum of the utilization of all tasks ran in the recently completed window.
- WALT "forgets" cpu utilization as soon as tasks are taken off of the runqueue, and thus the cpufreq governor can choose to drop frequency after just one window, potentially saving power

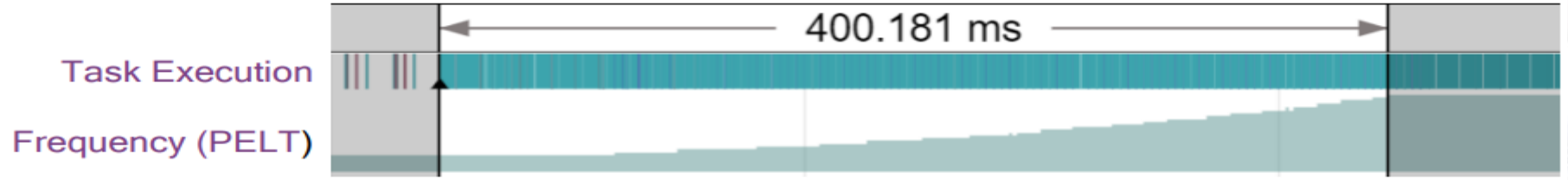
WALT vs PELT 1 of 3

- WALT has more dynamic demand/utilization signal than provided by PELT, which is subjected to the geometric series.
- WALT takes less time and PELT takes more time to detect a heavy task or re-classification of a light task as heavy. Thus the task migration to a higher capacity CPU is delayed. As example, it takes ~138msec for a task with 0 utilization to become a 95% utilization task.
- WALT ignores task when it goes to sleep, PELT decays the utilization of a task when it goes to sleep. As an example, a 100% utilization task would become 10% utilization task just after 100 msec sleep.
- WALT takes less time to ramp frequency up. PELT takes more time to build up the CPU utilization, thus delaying the frequency ramp up.
  - PELT's blocked utilization decay implies that the underlying cpufreq governor would have to delay dropping frequency for longer than necessary

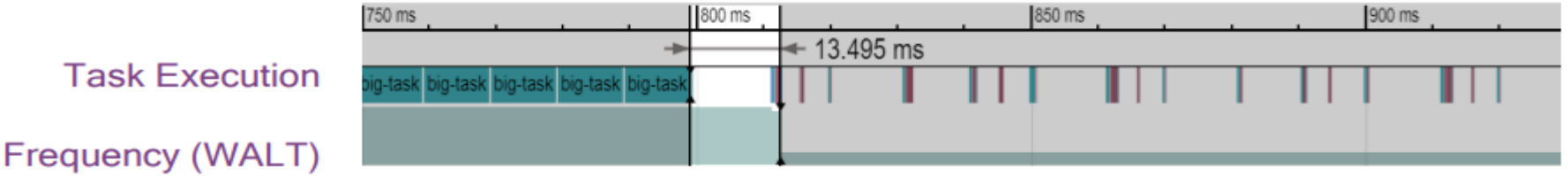
WALT vs PELT 2 of 3

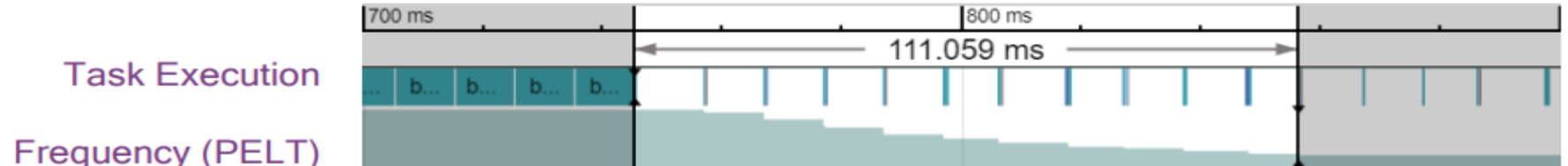
Frequency ramp up





Frequency ramp down





WALT vs PELT 3 of 3

	PELT	WALT
Load tracking	Load is accounted using a geometric series	Load is accounted with a policy that observes past *N* windows
Blocked load/utilization tracking	Load is decayed as part of a runqueue statistic when the task is blocked	Blocked load contribution is removed from runqueue sum/average statistics.
Blocked load restoration	Runqueue statistics include blocked load/utilization at all times	Load contribution is restored to RQ statistics when the task becomes runnable again.