Update on big.LITTLE scheduling experiments

Morten Rasmussen
Technology Researcher

Agenda

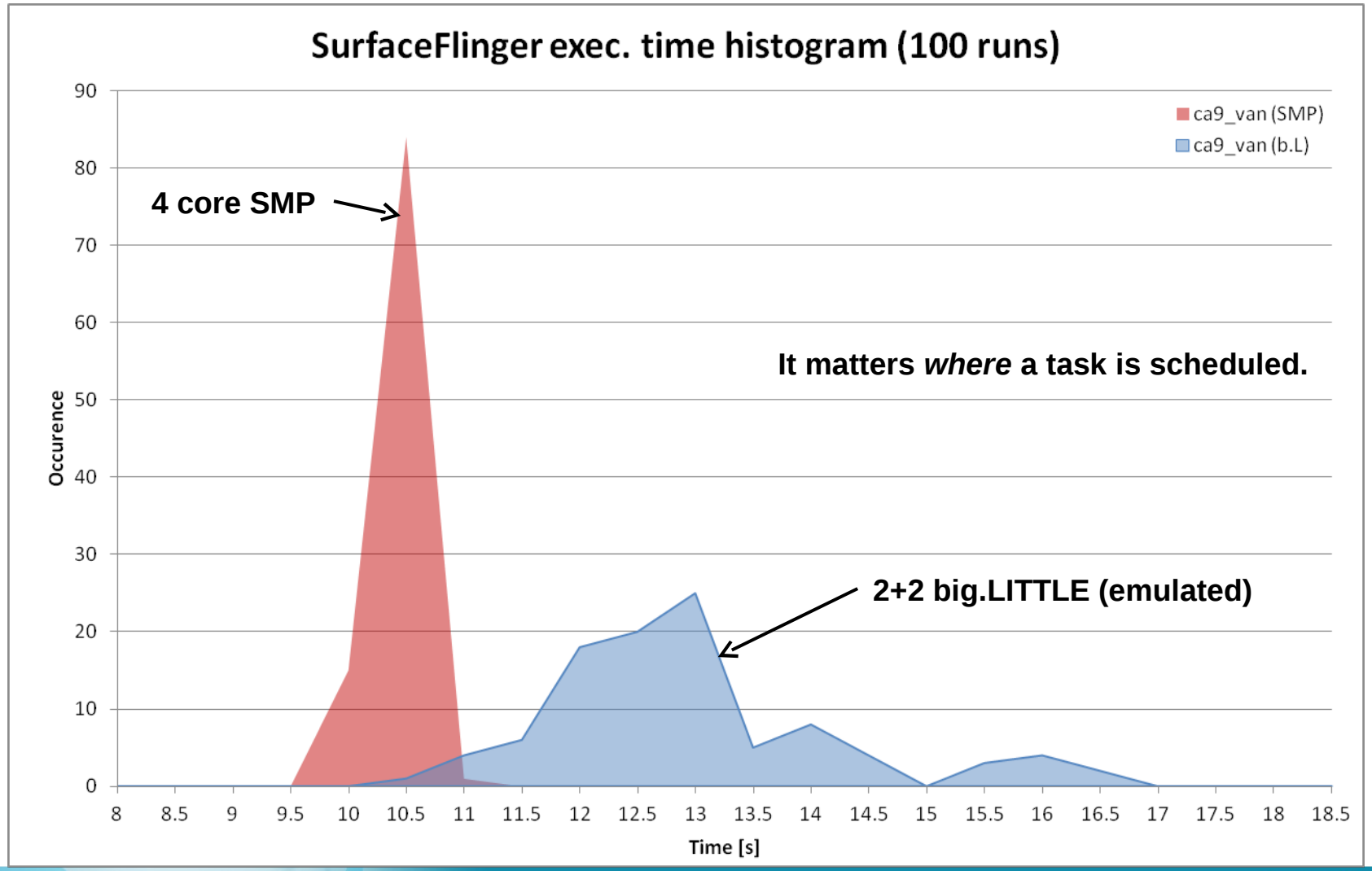
- Why is big.LITTLE different from SMP?
- Summary of previous experiments on emulated big.LITTLE.
- New results for big.LITTLE in silicon (ARM TC2).
- Next steps...

Why is big.LITTLE different from SMP?

- SMP:
  - Scheduling goal is to *distribute work evenly* across all available CPUs to get maximum performance.
  - If we have DVFS support we can even save power this way too.
- big.LITTLE:
  - Scheduling goal is to maximize power efficiency with only a modest performance sacrifice.
  - Task should be distributed *unevenly*. Only critical tasks should execute on big CPUs to minimize power consumption.
  - Contrary to SMP, it matters *where* a task is scheduled.

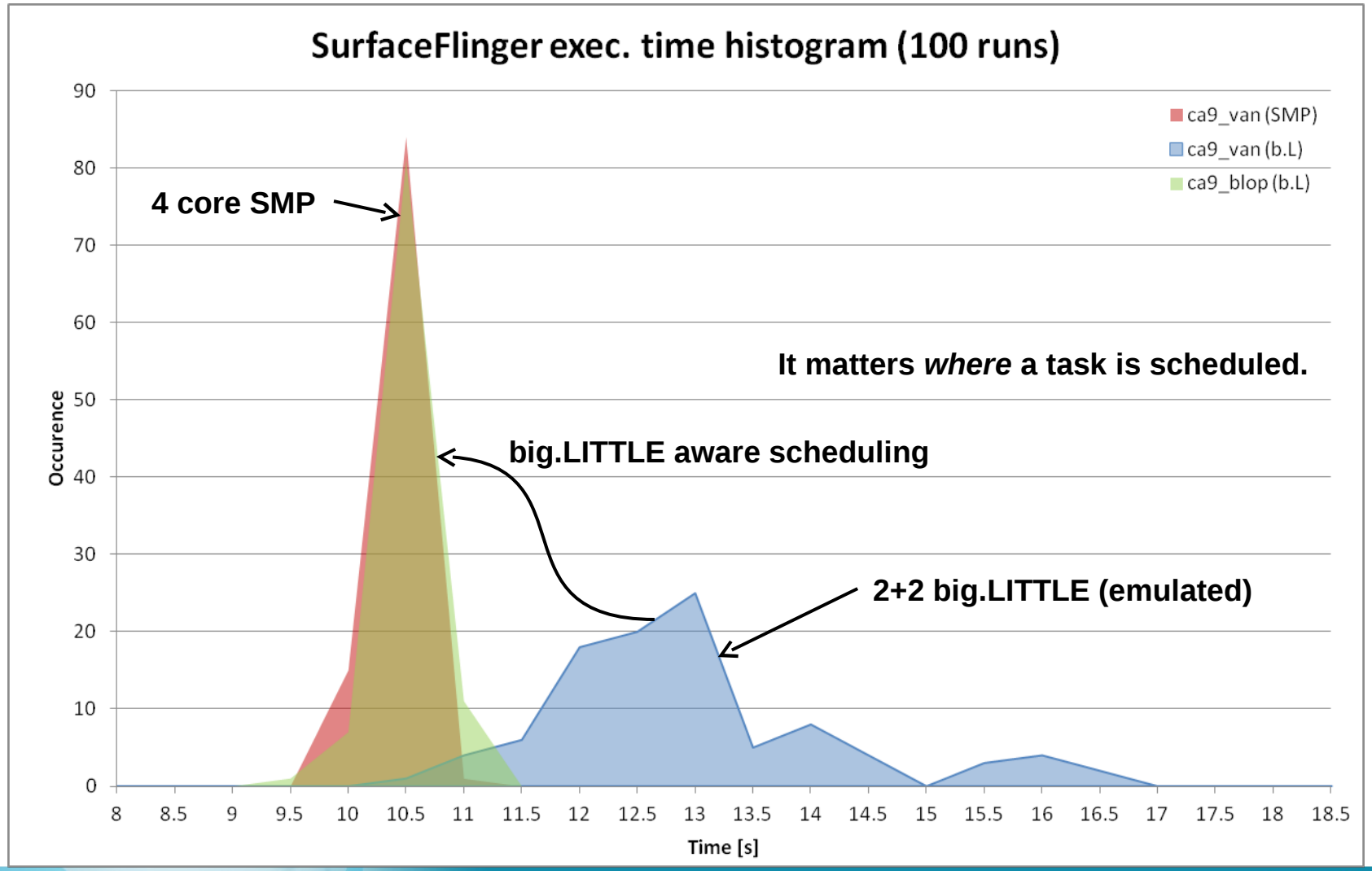
What is the (mainline) status?

- Example: Android UI render thread execution time.



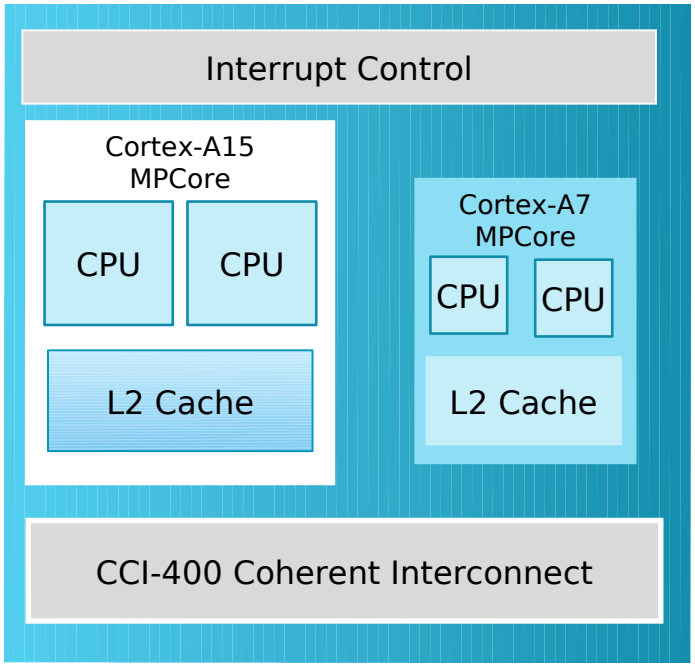
What is the (mainline) status?

- Example: Android UI render thread execution time.



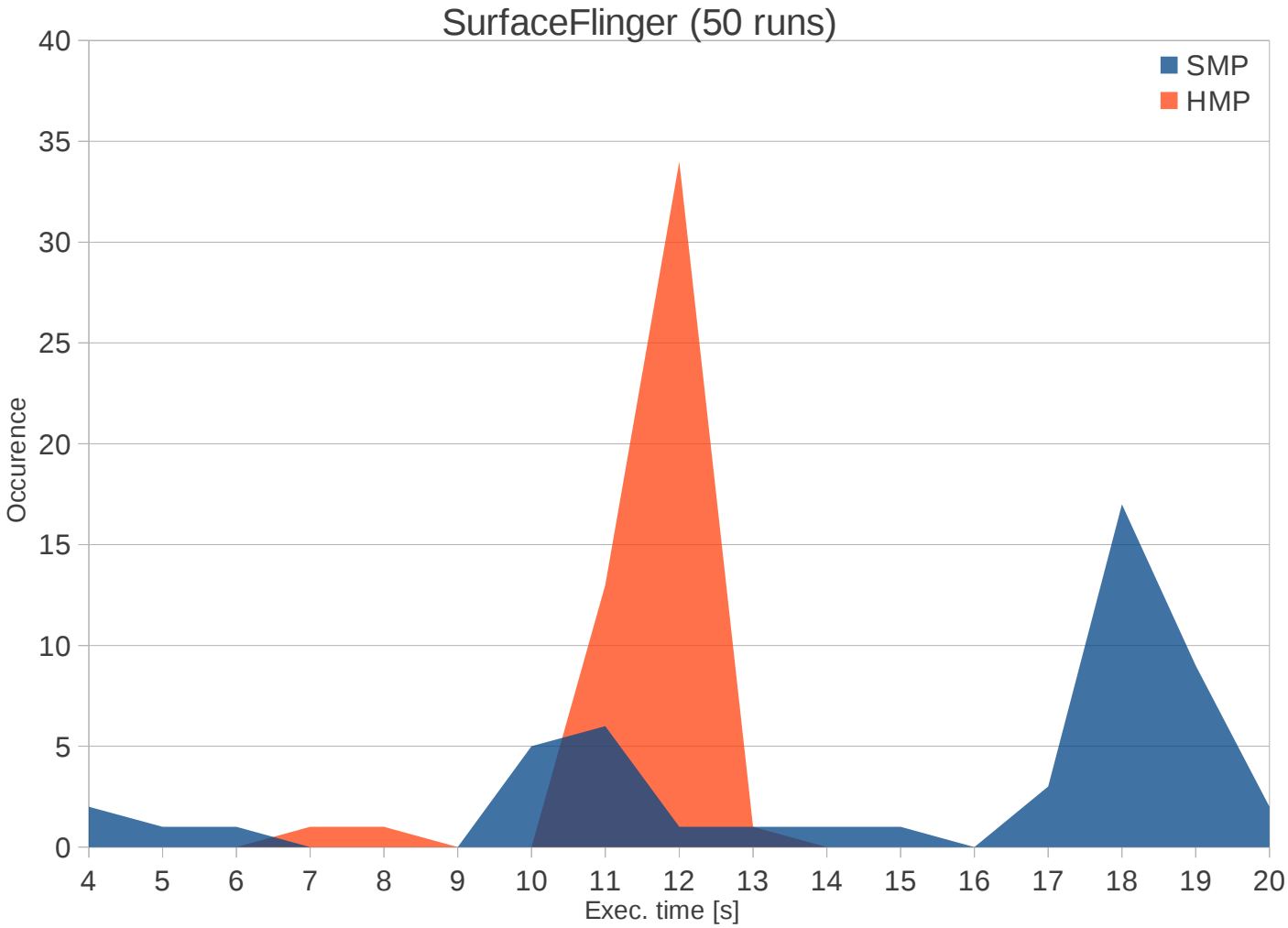
big.LITTLE hardware platform

- We are now in the process of investigating scheduling issues on real big.LITTLE hardware.
- ARM TC2 big.LITTLE test chip:
  - Two CPU clusters: 2x Cortex-A15 (big) + 3x Cortex-A7 (LITTLE)
  - Per-cluster L2 caches, cache coherent interconnect
  - No GPU
  - cpufreq support
  - cpuidle support
  - Linux SMP across all five cores



Running on real HW: ARM TC2

- Bbench on Android:



Mainline Linux Scheduler (CFS)

- We need proper big.LITTLE/heterogeneous system support in CFS.
  - Load-balancing is currently based on an expression of CPU load which is basically:

$$cpu_{load} = cpu_{power} \cdot \sum_{task} prio_{task}$$

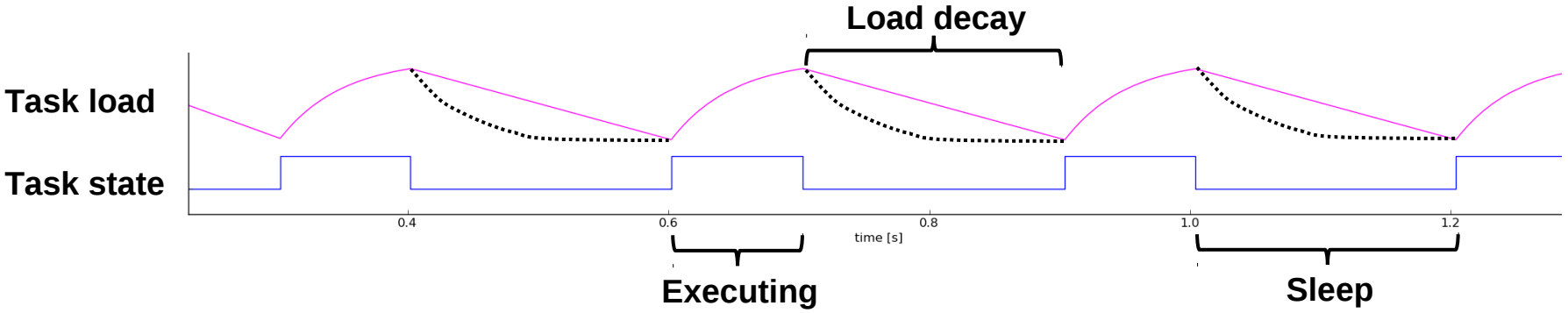
  - The scheduler does not know how much CPU time is consumed by each task.
  - The current scheduler can handle distributing tasks fairly evenly based on cpu_power for big.LITTLE system, but this is not what we want for power efficiency.
  - Embedded use cases focus mainly on responsiveness. It is therefore important that each task is scheduled on an appropriate cpu to get the best performance and power efficiency.

Tracking task load

- The load contribution of a particular task is needed to make an appropriate scheduling decision.
- We have experimented internally with identifying task characteristics based on the tasks' time slice utilization.
- Meanwhile, Paul Turner (Google) posted a RFC patch set on LKML with similar features.
  - LKML: https://lkml.org/lkml/2012/2/1/763
  - Focusing in improving fair group scheduling, but very useful for task placement on asymmetric systems.
  - Can potentially be used for aspects of power aware scheduling too.
  - This is now out in v2 (v3?). Mainline plans?

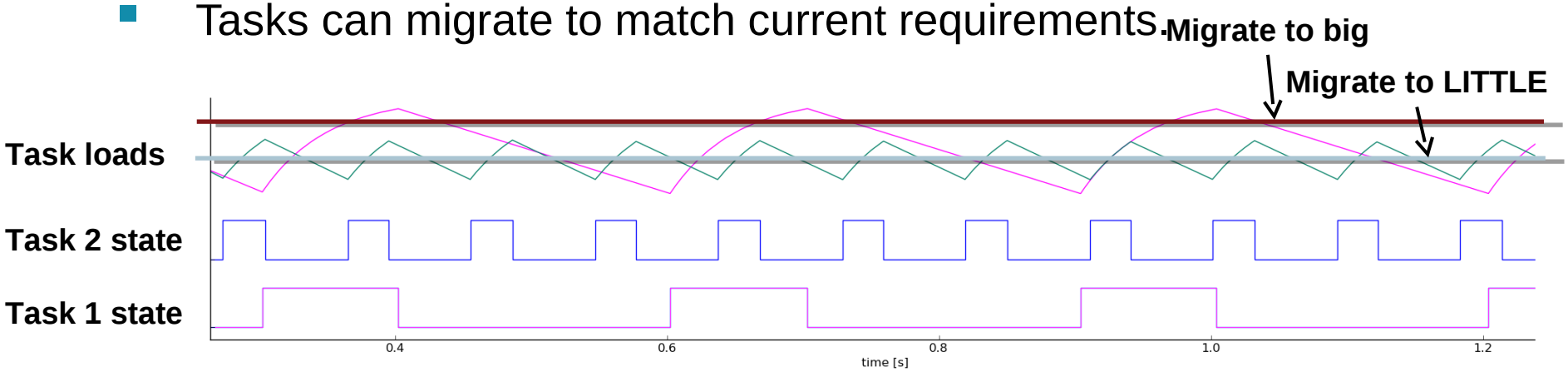
Entity load-tracking summary

- Tracks the time each task spends on the runqueue (executing or waiting) approximately every ms. Note that: $t_{runqueue} \geq t_{executing}$
- The contributed load is a geometric series over the history of time spent on the runqueue scaled by the task priority.
- Also task cpu usage and runqueue load.



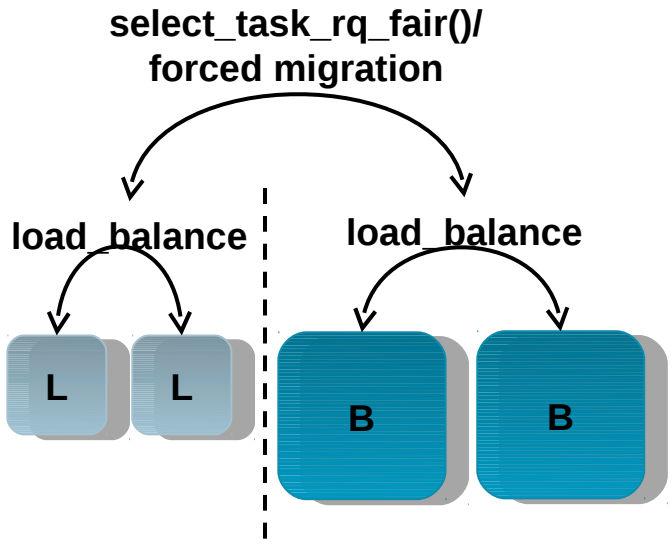
big.LITTLE scheduling: First stab

- Policy: Keep all task on little cores unless:
  1. The runqueue residency is above a fixed threshold, and
  2. The task priority is default or higher (nice ≤ 0)
- Goal: Only use big cores when it is necessary.
  - Frequent, but low intensity task are assumed to suffer minimally by being stuck on a little core.
  - High intensity low priority tasks will not be scheduled on big cores to finish earlier when it is not necessary.
  - Tasks can migrate to match current requirements.



Experimental Implementation

- Scheduler modifications:
  - Apply PJTs' load-tracking patch set.
  - Set up big and little sched_domains with no load-balancing between them.
  - select_task_rq_fair() checks task load history to select appropriate target CPU for tasks waking up.
  - Add forced migration mechanism to push of the currently running task to big core similar to the existing active load balancing mechanism.
  - Periodically check (run_rebalance_domains()) current task on little runqueues for tasks that need to be forced to migrate to a big core.
  - **Note:** There are known issues related to global load-balancing.

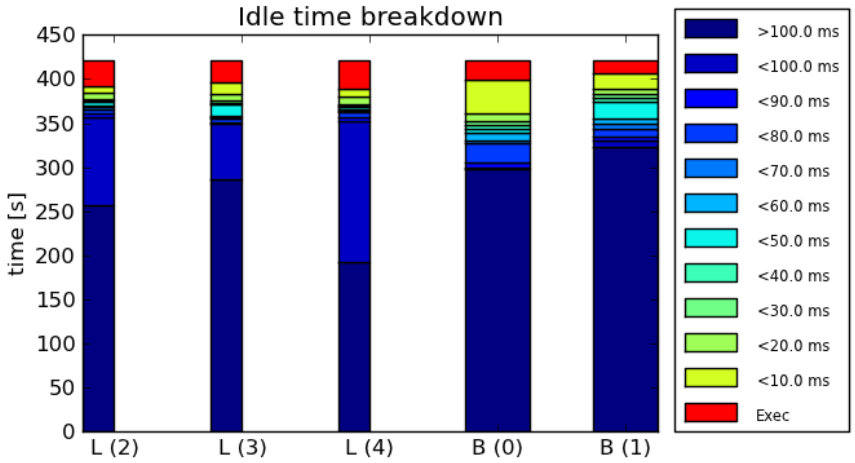


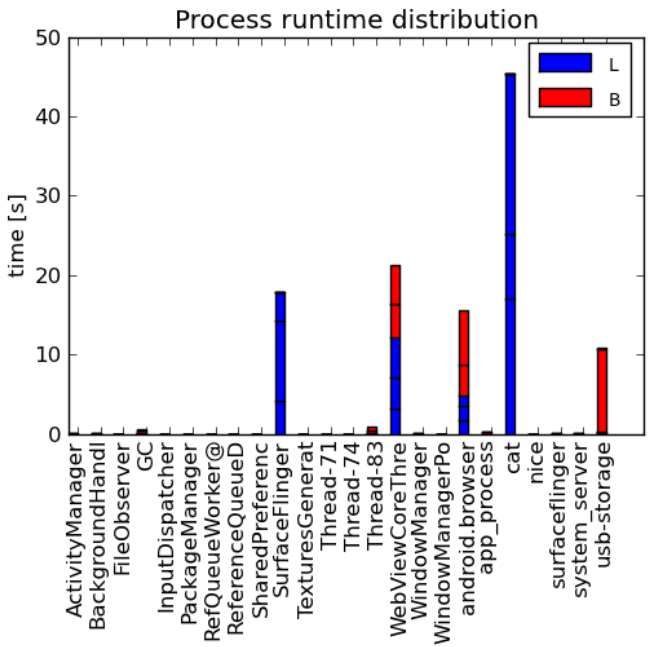
Forced migration latency:
~160 us on vexpress-a9
(migration->schedule)

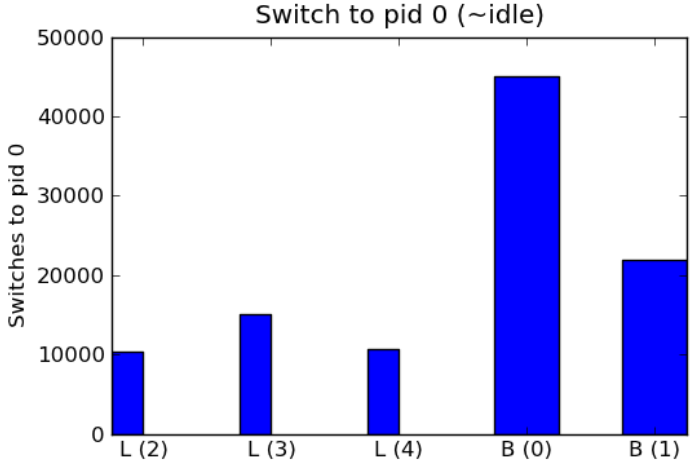
Example: Bbench on Android

- Filesystem: Android ICS (4.0)
- Browser benchmark
  - Renders a new webpage every ~50s using JavaScript.
  - Scrolls each page after a fixed delay.
  - Three main threads involved:
    - WebViewCoreThread: Webkit rendering thread.
    - SurfaceFlinger: Android UI rendering thread.
    - android.browser: Browser thread

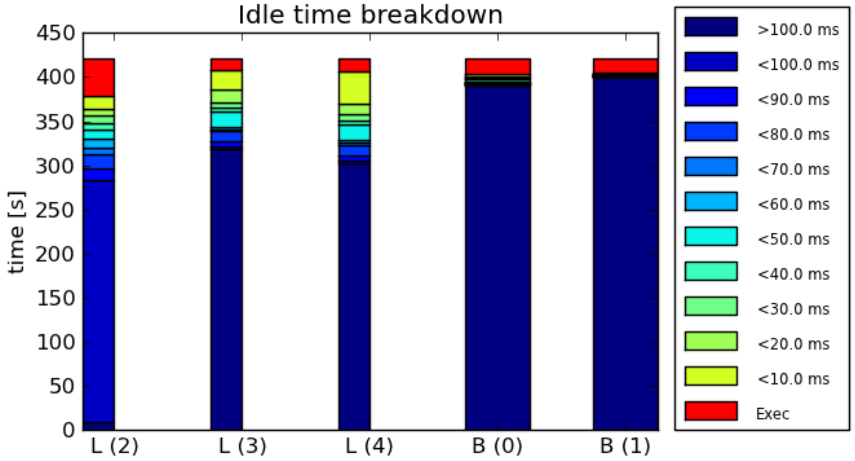
Bbench@TC2 SMP example analysis

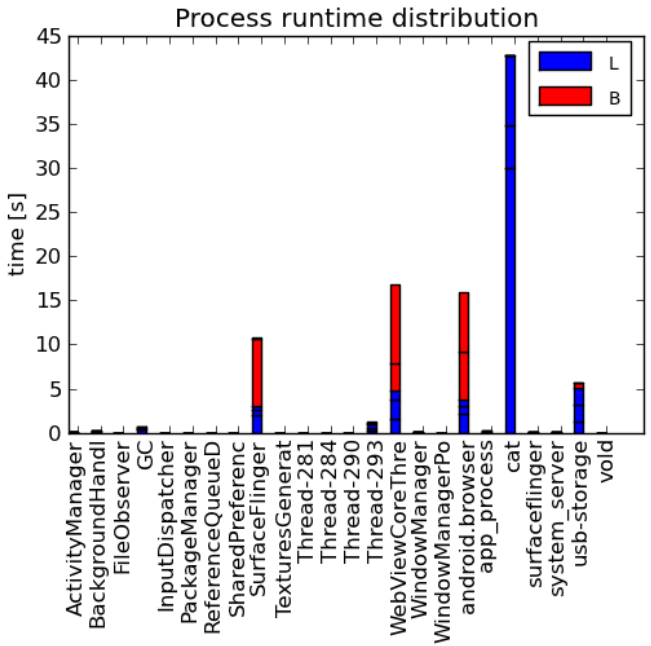


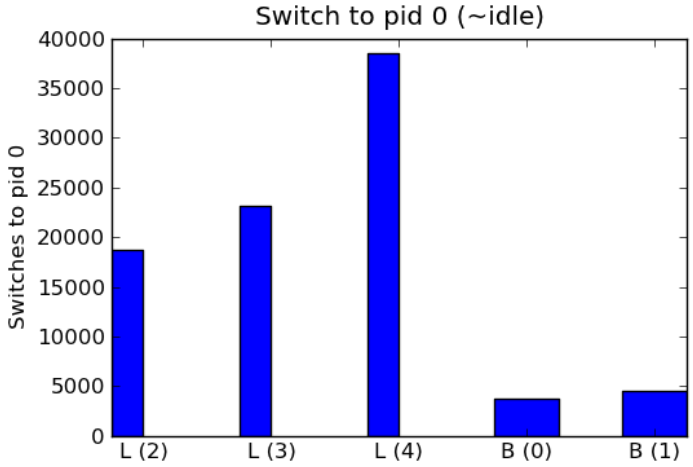




Bbench@TC2 HMP example analysis

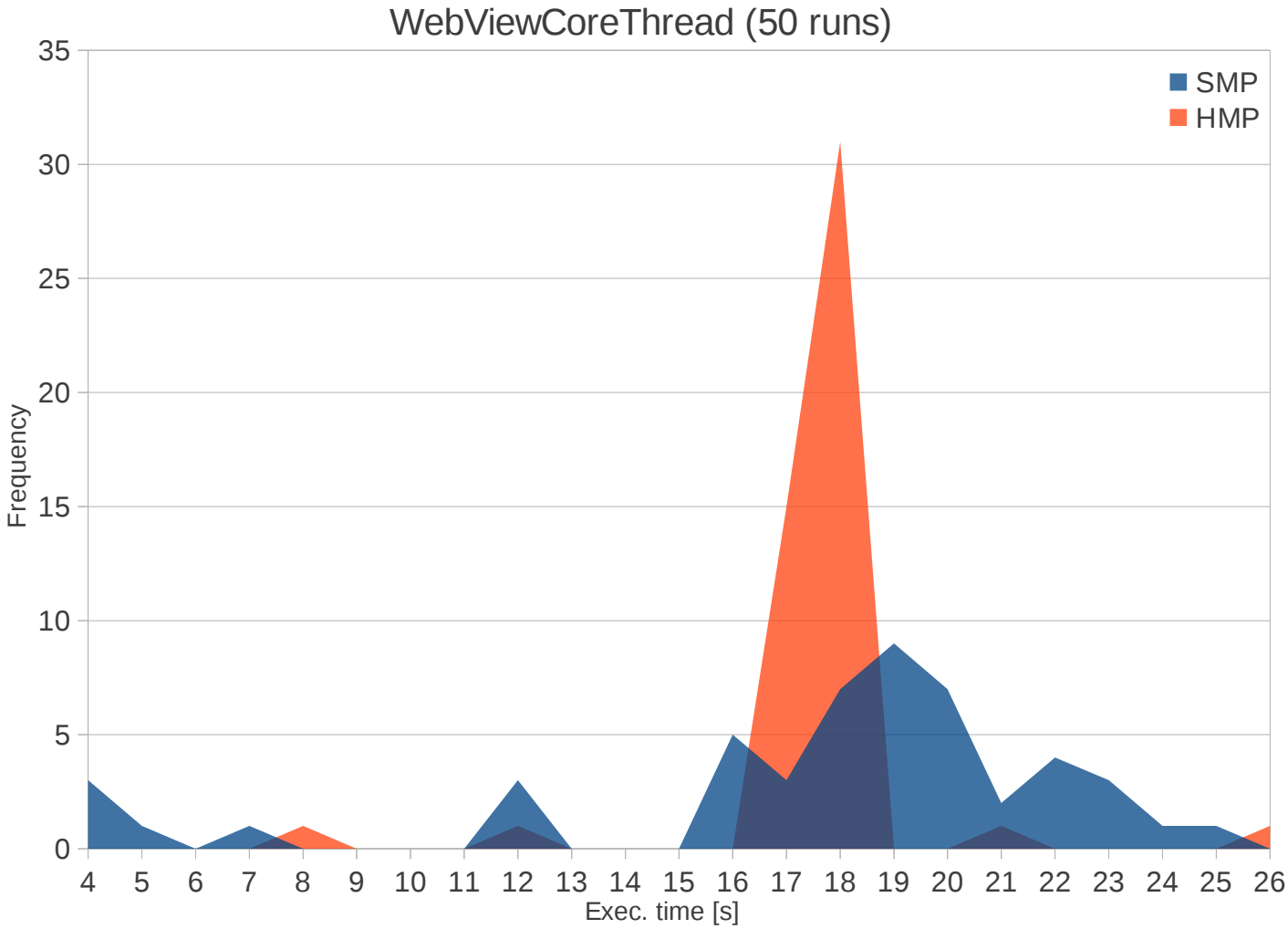






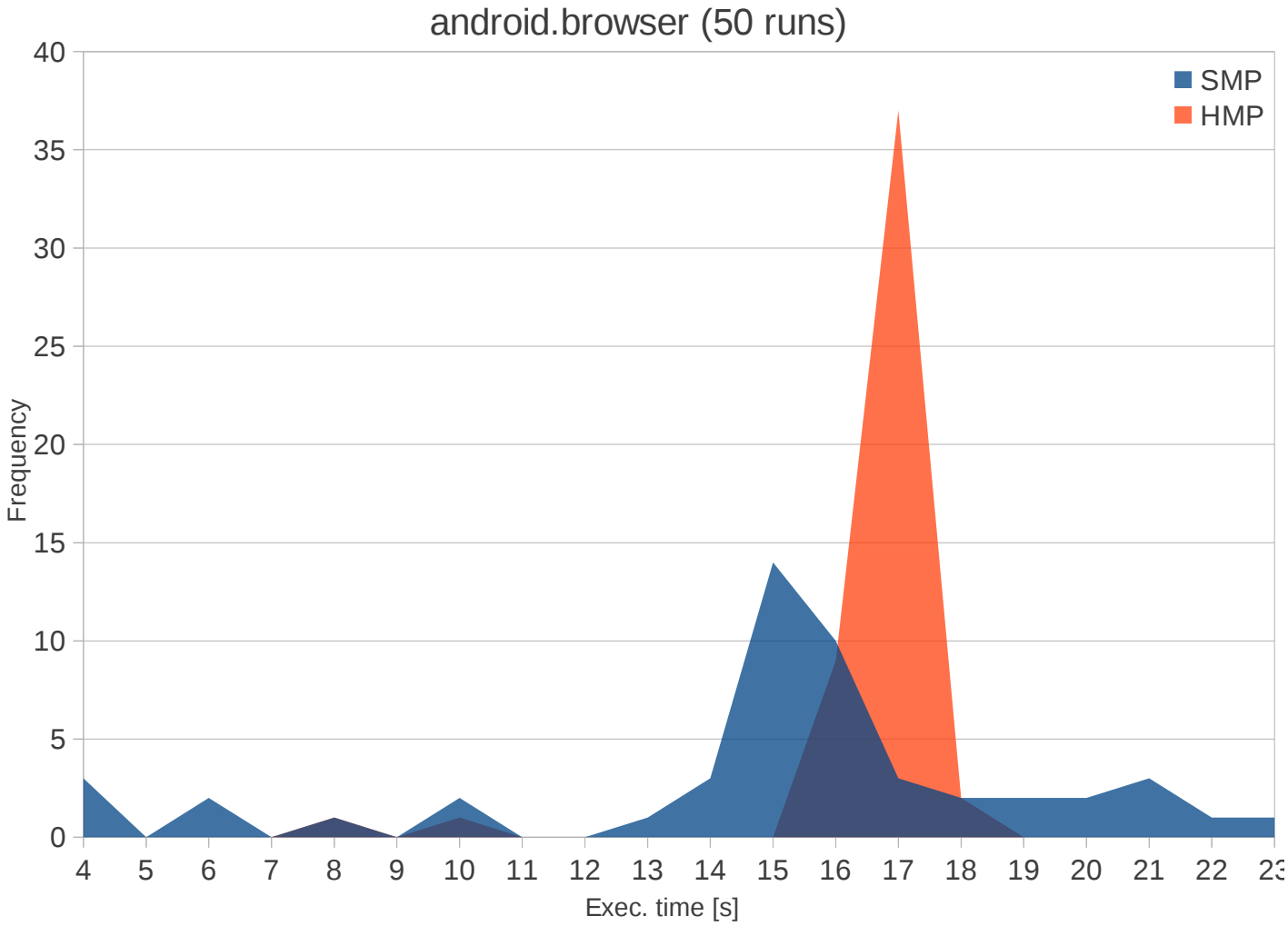
Bbench@TC2: WebViewCoreThread

- Bbench on Android:



Bbench@TC2: android.browser

- Bbench on Android:



Next step: Reimplementation of asymmetric task placement

- Experimental implementation disables the existing load balancing mechanism to override it.
- The current public (open Linaro repository) patch set is not meant for direct adoption, but serves as a tool for demonstration and evaluation.
- Ideally, a similar functionality should be integrated with the existing load balancer instead.
- Investigate the need for more control over task migrations (extra knobs). PJT's patches might need tuning knobs.
- Work on generalizing the patch set to support multiple cpu clusters is currently ongoing.
  - We only have two clusters (big.LITTLE) for testing.
  - Different target cluster selection policies for multi-cluster systems might be possible, but this is not our main focus for now.

Next step: Spread/Fill task placement

- Ongoing LKML discussions about power aware scheduling after SCHED_MC was removed.
- A spread/fill task distribution tuning knob is needed per cpu cluster for asymmetric systems like big.LITTLE.
  - For low leakage cpus spreading might be the best power/performance trade-off.
  - For high performance cpus filling might be better since leakage can be minimized.
- Task load could potentially be used for better cpu filling, but more investigation is needed.
  - The current implementation of tracked load might not be ideal as the individual task load is affected by the total cpu load.
  - A scale invariant task load metric might be needed, but is not trivial to define.

Next step: Integration with cpuidle

- Task load tracking gives the scheduler much more information about the tasks and the cpu load.
- Use this information to improve power aware scheduling in general. Not just for asymmetric systems.
- Example:
  - When waking up an idle cpu, select the one in the cheapest C-state.
- Related:
  - Selection of appropriate IRQ affinity. If cpu 0 is big, we need to be able to specify a different default target.

Next step: cpufreq intersections

- With task load tracking, the scheduler is in a good position to predict the cpu load every time a task is scheduled.
- Instead of waiting for cpufreq to figure out that the load has increased, it might be more efficient to drive/hint cpufreq from the scheduler.
- This would allow much more responsive and aggressive frequency scaling. Frequency transition latency is well below the schedule period on ARM TC2.
- Counterproductive scheduling behaviour can be avoided, e.g. the scheduler migrates tasks to another (idle) cpu before cpufreq has had a chance to increase the frequency.
- This applies to SMP system as well.
- For HMP, we also need to consider per cluster policies. Interactive/performance policy configuration on A7/A15 has shown good results in the lab for ARM TC2.

Questions?